Data Models for Conceptual Structures

Roger T. Hartley and Heather D. Pfeiffer

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003-8001, USA
email: rth@cs.nmsu.edu and hdp@cs.nmsu.edu

Abstract

A well-founded data model for Conceptual Structures can help in understanding issues of efficient implementations, definitional semantics and even syntax of proposed languages. This paper presents several useful data models of increasing complexity and applicability that can support Conceptual Structures. The models are presented in Haskell, a non-strict, strongly -typed functional language that allows easy definition of recursive data types, and also in straight XML using the proposed Schema standard that allows some of the same features. This paper then goes on to discuss how these models can lead to an efficient implementation in a schema-based XML version of an interchange format for Conceptual Graphs in comparison with CGIF. For the model presented in Haskell, a Haskell parser for CGIF syntax uses this model as a target, and this allows generation of XML output as a back-end to the parser. An XSLT style sheet that generates CGIF completes the cycle. All of the models are compared and contrasted.

Keywords: Data Models, Syntax, Semantics, Applications, Conceptual Structures and Knowledge Representation

1. Usefulness of Data Models

Conceptual structures, *CS*, as defined in Sowa's book [9], expresses declarative knowledge by implementing it as a connected multilabeled bipartite oriented graph. There exist a mapping from each of these graphs, *conceptual graphs,* to formulae in first-order logic. Each graph uses concept, relation, and actor type nodes, and links the total context together through the edges that connect them. Each label in a concept node consists of two fields, the *type* field and the *referent* field. The type field is an element of the set of concepts defined in a type lattice (see [6] for details). The referent field contains the individual specialization (if any) for the type field. Each label in a relation node consists of a single *relation* field. This relation field depicts the relationship between the adjoining concept nodes within the conceptual structure.

Sowa has shown how unknown objects (nodes with no individual field) can be computed by firing an *actor* node that corresponds to a function in standard logic. Actor nodes of this kind are connected to concept nodes with dashed lines in a graph to indicate that there is a functional procedure attached to this relation. However, even though these actors are procedural in nature, they are still declarative knowledge within the graph.

The Conceptual Graph Interchange Format (CGIF) was intended for easy interchange of conceptual graphs and until this present year was maintained as an ANSI standard [10] and updated by the community [7].

One view of CGIF is that it is a persistent version of the internal data structures that a processor for conceptual structures must maintain. This is not of course its only use, and was not the main focus of its beginnings as a means to transmit conceptual structures electronically. However, any processor that uses CGIF as its input-output format has to address the issue of what is the end result of parsing a CGIF file, and are there good and bad target data structures, or are they all equally good (or bad).

The authors contend that CGIF has a small number of natural data structures that lead to efficient parsing, and that an examination of the data models for these structures is useful for implementers of conceptual structure processors. Without such an examination, arbitrary choices can lead to in-efficient implementations and potential errors.

2. Data models

For the authors, the term 'data model' means the first half of an abstract data type (ADT) -- the definition of data types and structures that will support operations on those types. The operations are the second half of the ADT, but that is not part of the scope of this paper.

Many issues come to play when defining an ADT for a conceptual structures processor. Probably the most important is the efficiency of the operations in terms of their time and space requirements. Well-designed and well-defined data structures can certainly help in these respects, whereas poorly defined structures lead to inefficiencies. Again, this paper will not address these issues directly. Instead we shall concentrate on the data structures for the processors that use CGIF for input-output. This is not to say that a processor is constrained in its internal structures by CGIF, but it makes sense to examine the correspondence between CGIF and the appropriate data types with a view to minimizing the difficulties of parsing and generating CGIF syntax.

2.1. Haskell Data Model

We have chosen to make this discussion more formal by choosing a real language, Haskell, in which to define one of the data models. We could have used more traditional methods, such as those in the database world [[11], [2]] or in the object-oriented world [5], but a language like Haskell has several advantages:

1. It allows recursive data types without pointers, such as are essential for representing graphs.
2. It is strongly typed, which means that the language processor can check the definitions for errors.
3. We can write a parser for CGIF that has the chosen data model as a target. The existence of the Happy parser generator [4] was a bonus here.
4. We can add code to implement operations (not done here).
5. We can add code to generate different output forms -- here we have used XML.

2.1.1. Haskell Data Types

Haskell, like its earlier counterpart ML, has three main devices for defining types: the *constructor*, the *list*, and the *tuple*. Constructors allow types to be built as combinations of values (product types) or as enumerations of different alternative values, so a type for temperature can be defined as either Fahrenheit or Celsius:

```
data Temperature = Fahrenheit Float | Celsius Float
```

'Float' is a basic type; Fahrenheit and Celsius construct a Temperature value from that real number. So the expressions Fahrenheit 98.6 and Celsius 37.0 can both represent a value of type Temperature (it happens to be the same temperature). Constructors can have many arguments, and thus can generate compound values:

```
data Name String String String
```

can generate a name consisting of first, middle and last components.

It is also possible to name an existing compound type:

```
type NewType = (String,String,String)
```

This names the type consisting of a tuple of three strings.

List notation is similar to ML (and Prolog). Square brackets surround a comma-separated sequence of values all of the same type:

```
[1, 2, 3] is of type [Int], and [Fahrenheit 98.6, Celsius 37.0] is of type [Temperature]
```

These lists are single linked lists that have their origin in Lisp, and must be processed from left to right using operations head and tail. They are *not* arrays, although Haskell does also have arrays through extensions to the language.

The tuple is a finite sequence of values of possibly different types. Parentheses surround the comma-separated list of values:

```
(1, "normal", Fahrenheit 98.6) is of type (Int, String, Temperature)
(1, (2, 3), 4) is of type (Int, (Int, Int), Int)
```

2.2. Graph Data Models

We would also like to give data models generated by hand (without the help of a formal language) for conceptual graphs. Graphs can be modeled with many different representations [1], but as a basic definition of a graph we would like to use the following terminology [8]:

1. Denoted by G = <V, E>, where V is a finite set of vertices (nodes) and E is a set of edges, where an edge *ij* represents a connection between two nodes, *i* and *j*.
2. Defined as a set and a relation over that set, where every element of that set is a node and every relation is an edge.
3. If an edge has a direction then it is called an *arc*.

Using this base for a graph, we can now look at several different types of representations for our data models or ADTs.

2.2.1. Pointer Type

This representation produces an ADT for a general tree structure. That is one can traverse the tree from the top to the leaves moving in a breadth-first fashion. This is very space efficient because traversing the graph does not involve the creation of intermediate data structures.

There are however drawbacks to this representation [8]:

i) if nodes are not connected then more than one tree is produced for a single graph.
ii) since pointers that denote circularly defined values cannot be manipulated directly, constructing or traversing the graph cannot be achieved in a simple manner.

2.2.2. Adjacency List Type

This is a linear structure that holds every node in the graph together with its adjacent (connected with an edge) nodes. This is very useful for retrieving a node and all its edges, because they can be quickly retrieved using any starting node.

There are however drawbacks with this representation especially when the number of edges is dense:

i) it is not very space efficient (possible multiple copies of a node).
ii) takes extra time to insert a node into the graph at storage or modification time in order to maintain the added linkages in the lists.

2.2.3. Adjacency Matrix Type

As indicated in the type, this representation is a matrix (two-dimensional square array) of values of dimension nodes X nodes. The edges are indicated in the matrix by either the number 1 (true) as oppose to 0 (false) for no edge or a weighed value. This again is very useful for retrieving a node and all of its edges by just following down a column.

There are drawbacks with this representation specially when the number of edges is sparse:

i) it is not very space efficient , space = $|V|^2$
ii) it is not very space efficient when there are weighs on the edges, indicating when the edge is non-existent.

3. CGIF

The CGIF is a representation for conceptual graphs intended for transmitting CGs across networks and between IT systems that use different internal representations. The CGIF syntax ensures that all necessary syntactic and semantic information about a symbol is available before the symbol is used; therefore, all translations can be performed during a single pass through the input stream [10].

3.1. 'Natural' Models for CGIF

CGIF defines a graph as a set of concepts or relations, both of which are optional but there is an asymmetry. Relations may contain concepts (as 'arcs') but concepts may not contain relations. This leads to an inevitable, and unfortunate choice in the representation of any graph that has any complexity. The choice comes about because of the optional nature of both concepts and relations. Since a graph may contain disconnected concept nodes, we cannot base a graph purely on relations. Thus the simple graph:

```
(R1 [C1] [C2])
```

actually has three more alternative representations (ignoring the choices in ordering of the nodes):

```
[C1*x] (R1 ?x [C2])
[C2*y] (R1 [C1] ?y)
[C1*x] [C2*y] (R1 ?x ?y)
```

even though they are unnecessary, since the first is perfectly adequate. However, the graph:

```
[C1] (R1[C2])
```

has only one alternative (ignoring changes in the order of the nodes):

```
[C1] [C2*x] (R1 ?x)
```

since the unconnected node C1 must be kept separate.

We shall call the most compact form, with the least number of co-references, the 'minimum' form, and the one with the most co-references, the 'maximum' form. There are thus two 'natural' data models for CGIF, corresponding to these two extremes. When we consider that many of the operations on graphs are 'concept-first', then the maximum form would seem to lead to potential efficiencies in access. Both join and project are operations on concept nodes, so fast access to all the concept nodes in a graph would seem to be a good thing.

3.2. Haskell-based Data Model for CGIF

Our Haskell model for CGIF supports both minimum and maximum forms since there is no simple way to signal co-references between nodes, except as outlined above in the "pointer" model. If we neglect actors and special contexts for the moment, as well as lambda expressions in place of type labels, the model is:

```
type CG = ([CNode], [RNode])
type Label = String
data CNode = Concept Label Referent
data RNode = Relation Label InArcs OutArc
type InArcs = [CNode]
type OutArc = CNode
data Referent = Nil | Literal Literal | Graph CG
data Literal = LitString String | Name String | Marker String
```

Since Haskell can use the pointer model outlined above, a simple graph with one coreference link can be represented by an expression that includes two mentions of the same variable:

```
let sit = Concept "Sit" Nil in
   ([],[Relation "AGT" [sit] Concept "Cat" Literal Name "Fred",Relation "LOC" [sit] Concept
"Mat" Nil])
```

is the sentence "The cat Fred sitting on a mat". Since Haskell is non-strict, the expression does not imply that a copy of the expression bound to the variable sit by the let form is inserted in the two places it is referred to, but each will be evaluated as necessary. This is the 'lazy' evaluation style of Haskell.

It is also possible to explicitly include the coreference links in the data model:

```
type CG = ([CNode], [RNode])
type Label = String
data CNode = Concept Label Referent | DefConcept Label CoRef Referent | BoundConcept CoRef
type CoRef  = String
data RNode = Relation Label InArcs OutArc
type InArcs = [CNode]
type OutArc = CNode
data Referent = Nil | Literal Literal | Graph CG
data Literal = LitString String | Name String | Marker String
```

The sentence then becomes:

```
([],[Relation "AGT" [DefConcept "Sit" "x" Nil] Concept "Cat"  Literal Name "Fred",Relation
"LOC" [BoundConcept "x"] Concept "Mat" Nil])
```

The latter is closer to the ideas of coreference links in the CGIF definition.

The final data model, sufficient to capture all the information in the example files on the CGTools CD is:

```
type CG = ([CNode],[RNode],[ANode],[NNode])
type CoRef = String
data Node = CNode CNode | RNode RNode | ANode ANode | NNode NNode
data CNode = Concept String CoRef Referent | CLambda AList CG CoRef Referent
data RNode = Relation String [CNode] | RLambda AList CG [CNode]
data ANode = Actor String [CNode] [CNode]
data Type = TypeLabel String | Lambda AList CG
data NNode = Negation CG
data Quantifier = QuantifierNum Int | QuantifierId String
data Referent = Nil | Literal Literal | Graph CG | Quantifier Quantifier |
       LiteralGraph Literal CG | QuantifierGraph Quantifier CG
data Literal = LitString String | Name String | Marker String |
       Set [Literal] | List [Literal] | Array [Literal] |
       Tuple [Literal]
```

This model includes actors, lambda expressions, negation contexts, quantifiers and the CP style of collections [5].

3.3. A Haskell-based Parser for CGIF

The existence of the ‘Happy’ Haskell parser generator [4] led to a series of trials with the aim of parsing CGIF with different data models as targets. A total of twelve different versions were tried, and the degree of difficulty noted. All were successful without having to write too much extra code to ‘squeeze’ the data from a CGIF file into the target data model. Coreferences can be handled either by the pointer model or the explicit model and the minimum or maximum forms can be targeted.

The grammar[1] used for input to the Happy parser generator is very close to CGIF but eliminates some of the difficulties and cleans up a little of what was left. The more complex models enabled all five levels of the CGIF examples on the CGTools workshop CD to be parsed into Haskell data, without any loss of information. This has been verified by generating XML version of the parsed file that can be turned back into CGIF syntax with an XSL stylesheet, and comparing the newly generated version with the original. This work will be presented below.

4. XML Schema-based models

4.1. Generating XML from the Haskell Model

The last data model in section 3 was used as the source of an XML schema-based model for conceptual structures. The Schema generation tool in XMLSpy was used for this work since it guarantees that the resulting schema is valid. Appendix A presents this schema. Each data type in the Haskell model becomes a complex type in the XML version. XML actually has a little more flexibility than Haskell data types when it comes to ordering of components of compound types, and this was taken advantage of. Coreferences are handled by an optional attribute on a concept tag. Thus the “cat on a mat” example becomes:

```
<CGFile>
   <CG>
      <RNode>
         <Label>AGT</Label>
         <OutArc CoRef="x">
            <Label>Sit</Label>
         </OutArc>
         <InArc>
            <Label>Cat</Label>
            <Referent>
               <Literal>
                  <Name>Fred</Name>
               </Literal>
            </Referent>
         </InArc>
      </RNode>
```

[1] Full grammar may be viewed at web site: http://www.cs.nmsu.edu/~hdp/XML/NMSU/happy_grammar.txt

```
      <RNode>
         <Label>LOC</Label>
         <OutArc CoRef="x"/>
         <InArc>
            <Label>Mat</Label>
         </InArc>
      </RNode>
   </CG>
</CGFile>
```

Haskell code was written to generate this XML output from the data generated by the Haskell parser. This was made simple by the monad style of I/O allowed in Haskell, that turns a tricky task in a functional language into something like Basic programming.

4.2. Generating XML for Graph Models

As discussed in section 2.2., many different types of graph data models can be produced by hand besides using the Haskell language to produce a data model. We have created data models for some of the graph types discussed in the previous section. The adjacency list and matrix types are efficient for retrieving concepts and their links for sparse and dense node sets, respectively. However, when they are converted to XML, by hand, the space problem seen in the abstract data model is carried over into the XML schema produced. For the adjacency list data model, the XML for the actual conceptual graph becomes five lists with the following basic structure[2]:

```
<cgraph>
   <con-list>
      <conceptref>
  </con-list>
  <RC-list>
      <pair>
         <relationref>
         <conceptref>
      </pair>
  </RC-list>
  <CR-list>
      <pair>
         <conceptref>
         <relationref>
      </pair>
  </CR-list>
  <AC-list>
      <pair>
         <actorref>
         <conceptref>
      </pair>
  </AC-list>
  <CA-list>
      <pair>
         <conceptref>
         <actorref>
      </pair>
  </CA-list>
</cgraph>
```

[2] Full XML Schema can be seen at the following web site: http://www.cs.nmsu.edu/~hdp/XML/NMSU/listcgif.txt

The adjacency matrix data model would have the same space problem, but even worse if there were a sparse number of edges in the model.

We did discover that if we hand coded the XML schema to be more like the graph pointer data type discussed in section 2.2.1, space was not so much of a problem. However, to make retrieval of the concept nodes faster and more efficient, we did not use a tree structure, but instead used the tuple data structure from database technology [11].

This produced a very similar data model to both the Haskell and adjacency list data models when it came to the internal node structure, but a much more efficient data model for the actual conceptual graph. The final XML Schema can be seen in Appendix B that was constructed using XMLSpy.

5. Completing the Cycle: An XSL Style Sheet for CGIF

In order to verify the data models used the authors decided to 'complete the cycle' and generate CGIF syntax from the XML output of the Happy-generated parser. (CGIF -> Haskell data -> XML -> CGIF). This is most easily accomplished with an XSL stylesheet, rather than using an XML parser. The stylesheet contains commands for transforming XML input into a desired output form with the help of the parser built into the Internet Explorer browser (we used version 6 since it has the most up-to-date version of the XML parser and XSL transform engine built into it.) XMLSpy also has an IE6-compatible transform engine that we used for testing. The style sheet is unremarkable except for its handling of whitespace which is something of a black art. When in readable form, the style sheet produces unwanted whitespace in the output[3].

6. Conclusions

The authors have found that exploring different data models that can support conceptual structures separately from a detailed implementation has been very instructive. The range of models described in this paper covers many issues in defining and implementing processors for conceptual structures. The use of Haskell as a data definition language means that different data models can be compared easily, and then used as the target of a parser for CGIF. Design of an XML schema is then simplified since the raw material of type definitions had already been worked through. A comparison with an existing implementation, designed in the traditional way, was also made easier. The authors plan to improve their implementations in the future with a study of data models such as the one presented here.

Appendix A: The XML Schema produced from the Haskell data model

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.2 (http://www.xmlspy.com) by Roger Hartley (New Mexico State University) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
	<xs:element name="CGFile">
		<xs:annotation>
			<xs:documentation>Root Element</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="CG" minOccurs="0" maxOccurs="unbounded"/>
```

[3] Full XSL Style Sheet may be viewed at web site: http://www.cs.nmsu.edu/~hdp/XML/NMSU/xsl_stylesheet.txt

```
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="CG" type="CGType"/>

<xs:complexType name="NodeType">
    <xs:choice>
        <xs:element name="Label" minOccurs="0">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="xs:string"/>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="LambdaExpression">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="Param" maxOccurs="unbounded">
                        <xs:complexType>
                            <xs:complexContent>
                                <xs:restriction base="CNodeType">
                                    <xs:sequence>
                                        <xs:element name="Label">
                                            <xs:complexType>
                                                <xs:simpleContent>
                                                    <xs:extension base="xs:string"/>
                                                </xs:simpleContent>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:sequence>
                                </xs:restriction>
                            </xs:complexContent>
                        </xs:complexType>
                    </xs:element>
                    <xs:element ref="CG"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:choice>
    <xs:attributeGroup ref="Position"/>
</xs:complexType>
<xs:complexType name="CGType">
    <xs:sequence>
        <xs:element name="CNode" type="CNodeType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="RNode" type="RNodeType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="ANode" type="ANodeType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="NNode" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="CG"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="CNodeType">
    <xs:complexContent>
        <xs:extension base="NodeType">
            <xs:sequence>
                <xs:element name="Referent" minOccurs="0">
                    <xs:complexType>
                        <xs:all>
                            <xs:element ref="Literal" minOccurs="0"/>
                            <xs:element ref="CG" minOccurs="0"/>
                            <xs:element name="Quantifier" minOccurs="0">
                                <xs:complexType>
                                    <xs:choice>
                                        <xs:element name="Number" type="xs:positiveInteger"/>
                                        <xs:element name="Type" type="xs:string"/>
                                    </xs:choice>
                                </xs:complexType>
                            </xs:element>
                        </xs:all>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <xs:attribute name="CoRef" type="xs:string" use="optional"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="RNodeType">
    <xs:complexContent>
        <xs:extension base="NodeType">
            <xs:sequence>
                <xs:element name="OutArc" type="CNodeType"/>
                <xs:element name="InArc" type="CNodeType" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="ANodeType">
    <xs:complexContent>
        <xs:restriction base="NodeType">
            <xs:sequence>
                <xs:element name="Label">
                    <xs:complexType>
                        <xs:simpleContent>
                            <xs:extension base="xs:string">
                                <xs:attribute name="XPos" type="xs:short" use="optional"/>
                                <xs:attribute name="Ypos" type="xs:short" use="optional"/>
                            </xs:extension>
                        </xs:simpleContent>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Input" type="CNodeType" maxOccurs="unbounded"/>
                <xs:element name="Output" type="CNodeType" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:restriction>
    </xs:complexContent>
</xs:complexType>
<xs:element name="Literal">
    <xs:complexType>
        <xs:choice>
            <xs:element name="Marker" type="xs:positiveInteger"/>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="LitString" type="xs:string"/>
            <xs:element name="Set">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="Literal" minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="Array">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="Literal" minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="List">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="Literal" minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="Tuple">
                <xs:complexType>
                    <xs:sequence>
```

```
                            <xs:element ref="Literal" minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:choice>
        </xs:complexType>
    </xs:element>
    <xs:attributeGroup name="Position">
        <xs:attribute name="XPos" type="xs:positiveInteger" default="0"/>
        <xs:attribute name="YPos" type="xs:positiveInteger" default="0"/>
    </xs:attributeGroup>
</xs:schema>
```

Appendix B: The XML Schema produced by hand as an efficient data model

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.0 U (http://www.xmlspy.com) by Heather D. Pfeiffer (NMSU / Dept. CS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="cgraph">
        <xs:annotation>
            <xs:documentation>Conceptual Graph - Adjacency matrix like data structure, but in tuples</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:all>
                <xs:element name="con-list" type="conceptType"/>
                <xs:element name="RCC-tuple" type="RCC-tupleType" minOccurs="0"/>
                <xs:element name="ACC-tuple" type="ACC-tupleType"/>
            </xs:all>
            <xs:attribute name="uniquegraph" type="xs:ID" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="c-in-out" type="c-in-outType">
        <xs:annotation>
            <xs:documentation>concept list that has pairs of concepts in and out, must have both sides</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="c-posin-out" type="c-posin-outType">
        <xs:annotation>
            <xs:documentation>concept list in pairs with concepts in and out, does not have to have a concept in, only out</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:complexType name="ACC-tupleType">
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="actor"/>
            <xs:element ref="c-in-out"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="c-posin-outType">
        <xs:all maxOccurs="unbounded">
            <xs:element ref="concept" minOccurs="0"/>
            <xs:element ref="concept"/>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="RCC-tupleType">
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="relation"/>
            <xs:element ref="c-posin-out"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="c-in-outType">
        <xs:all maxOccurs="unbounded">
            <xs:element ref="concept"/>
            <xs:element ref="concept"/>
        </xs:all>
    </xs:complexType>
    <xs:element name="cgs">
        <xs:annotation>
            <xs:documentation>Set of Conceptual Graphs in a file</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
            <xs:element ref="cgraph"/>
            <xs:element name="period" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="concept">
    <xs:annotation>
        <xs:documentation>Concept Node - gives the conceptual type information</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="referent" minOccurs="0"/>
            <xs:element ref="typelabel" minOccurs="0"/>
            <xs:element ref="cgraph" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="uniquecon" type="xs:ID" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="relation">
    <xs:annotation>
        <xs:documentation>Relation Node - gives a conceptual relationship between concept types</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:all>
            <xs:element ref="typelabel"/>
        </xs:all>
        <xs:attribute name="uniquerel" type="xs:ID" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="actor">
    <xs:annotation>
        <xs:documentation>Actor Node - gives a conceptual relationship with a functional element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:all>
            <xs:element ref="typelabel"/>
            <xs:element name="function" type="xs:ENTITY"/>
        </xs:all>
        <xs:attribute name="uniqueact" type="xs:ID" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="referent">
    <xs:complexType>
        <xs:all>
            <xs:element name="literal">
                <xs:complexType>
                    <xs:all>
                        <xs:element ref="litstring" minOccurs="0"/>
                        <xs:element name="number" type="xs:integer" minOccurs="0"/>
                    </xs:all>
                </xs:complexType>
            </xs:element>
            <xs:element name="locator">
                <xs:complexType>
                    <xs:all>
                        <xs:element name="name" type="xs:Name" minOccurs="0"/>
                        <xs:element name="marker" type="xs:positiveInteger" minOccurs="0"/>
                        <xs:element name="identifier" type="xs:ID" minOccurs="0"/>
                    </xs:all>
                </xs:complexType>
            </xs:element>
            <xs:element name="quantifier">
                <xs:complexType>
                    <xs:all>
                        <xs:element name="qmarker" type="xs:positiveInteger" minOccurs="0"/>
                        <xs:element name="liststring" minOccurs="0">
                            <xs:complexType>
```

```
                                        <xs:sequence maxOccurs="unbounded">
                                            <xs:element ref="litstring"/>
                                        </xs:sequence>
                                    </xs:complexType>
                                </xs:element>
                            </xs:all>
                        </xs:complexType>
                    </xs:element>
                </xs:all>
            </xs:complexType>
        </xs:element>
        <xs:element name="typelabel">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="xs:string"/>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="litstring" type="xs:string"/>
        <xs:complexType name="conceptType">
            <xs:sequence maxOccurs="unbounded">
                <xs:element ref="concept"/>
            </xs:sequence>
        </xs:complexType>
    </xs:schema>
```

References

[1] Even, S. (1979) Graph Algorithms. Rockville, MD: Computer Science Press.

[2] Lewis, P.M., A. Bernstein and M. Kifer (2002) Databases and Transactions Processing: An Application-Oriented Approach. Reading, MA: Addison-Wesley.

[3] Marlow, S. (2001) The Happy Parser Generator, Version 1.11. [Access Online:September 2001], URL: http://www.haskell.org/happy/.

[4] Page-Jones, M. (2000) Fundamentals of Object-Oriented Design in UML. New York, NY: Dorset House Publishing and New York, NY: Addison-Wesley.

[5] Pfeiffer, H.D. and R.T. Hartley (1990) Set Representation and Processing Additions to Conceptual Programming, in Proc. of the Fifth Annual Workshop on Conceptual Graphs, Boston, MA, pp. A.15-1 – 10.

[6] Pfeiffer, H.D. and R.T. Hartley (1992) Temporal, Spatial, and Constraint Handling in the Conceptual Programming Environment, CP, in Journal for Experimental and Theoretical AI Vol. 4 No. 2, pp.167-182.

[7] Pfeiffer, H.D. and R.T. Hartley (2001) ARCEdit- CGEditor, in CGTools Workshop Proceedings in connection with ICCS2001, Stanford, CA. [Access Online:July 2001], URL: http://www.cs.nmsu.edu/~hdp/CGTools/proceedings/index.html.

[8] Rabhi, F. and G. Lapalme (1999) Algorithms: a functional programming approach. Reading, MA: Addison-Wesley.

[9] Sowa, J.F. (1984) Conceptual Structures. Reading, MA: Addison-Wesley.

[10] Sowa, J.F. et al (2001) Conceptual Graph Standard, American National Standard NCITS.T2/ISO/JTC1/SC 32 WG2 M 000. [Access Online:April 2001], URL: http://www.bestweb.net/~sowa/cg/cgstand.htm.

[11] Ullman, J.D. (1989). Principles of Database and KnowledgeBase Systems, Volumes I,II. Rockville, MD: Computer Science Press.