An introduction to Conceptual Graphs

POLOVINA, S. <http://orcid.org/0000-0003-2961-6207>

An Introduction to Conceptual Graphs

Simon Polovina

Culture, Communication and Computing Research Institute (C3RI)
Faculty of Arts, Computing, Engineering & Sciences
Sheffield Hallam University, UK S1 1WB
s.polovina@shu.ac.uk

Abstract. This paper provides a lucid introduction to Conceptual Graphs (CG), a powerful knowledge representation and inference environment that exhibits the familiar object-oriented features of contemporary enterprise and web applications. An illustrative business case study is used to convey how CG adds value to data, including inference for new knowledge. It enables newcomers to conceptual structures to engage with this exciting field and to realise "Conceptual Structures: Knowledge Architectures for Smart Applications", the theme of the 15th Annual International Conference on Conceptual Structures (ICCS 2007, www.iccs2007.info).

1 Introduction

Conceptual Graphs (CG, www.conceptualgraphs.org) provide a powerful knowledge representation and inference environment, whilst exhibiting the familiar object-oriented and database features of contemporary enterprise and web applications. CG capture nuances in natural language whilst being able to be implemented in computer software. CG were devised by Sowa from philosophical, psychological, linguistic, and artificial intelligence foundations in a principled way [8, 9]. Hence CG are particularly attractive as they are built upon such a strong theoretical and wide-ranging base.

There is an active CG community, evidenced by the annual International Conferences on Conceptual Structures (ICCS, www.iccs.info), now in its 15th year (ICCS 2007, www.iccs2007.info), not to mention the annual CG workshops beforehand (www.conceptualstructures.org/confs.htm). There is also the CG discussion list (cg@conceptualgraphs.org). Its participants happily support newcomers to CG e.g. in answering queries; www.conceptualgraphs.org provides information on how to join this list, as well as a comprehensive catalogue of software CG tools.

CG is core to the ISO Common Logic standard (http://cl.tamu.edu/)[1]. The CG community has furthermore grown to embody a wider notion of Conceptual Structures (CS, www.conceptualstructures.org, and the title of Sowa's seminal 1984 text [9]). This is typified by the large scale and valued contributions that Formal Concepts Analysis (FCA, www.upriss.org.uk/fca) brings to ICCS each year [4].

[1] Assigned to WG2 (Metadata) under SC32 (Data Interchange) of ISO/IEC JTC1.

A strong case therefore exists for bringing an awareness of CG to an even wider community. Through an illustrative business case study, the following explication of CG [2] aims to achieve this objective so that many more researchers and industry professionals can realise the benefits of CG and, as the theme of ICCS 2004 highlighted, put them to work [10].

2 Concepts and Relations

CG are based upon the following general form:

This may be read as: "*The* relation *of a* Concept-1 *is a* Concept-2". The direction of the arrows assists the direction of the reading. If the arrows were pointing the other way, then the reading would be the same except that Concept-1 and Concept-2 would exchange places (i.e. "*The* relation *of* Concept-2 *is a* Concept-1").

As an alternative to the above 'display' form[3], the graphs may be written in the following 'linear' text-based form:

```
[Concept_1] -> (relation) -> [Concept_2].
```

The full stop '.' signals the end of a particular graph. Consider the following example:

```
[Funding_Request] -> (initiator) -> [Employee].
```

This example will form a part of an illustrative case study about requests for funding new business projects in the fictitious enterprise 'P-H Co.'. The example graph reads as "The initiator of a funding request is an employee". This may create readings that may sound long-winded or ungrammatical but is a useful mnemonic aid in constructing and interpreting any graph. It is easier to state, "An employee initiates a funding request".

Concepts can have *referents*, which refers to a particular instance, or individual, of that concept[4]. For example, consider the concept:

```
[Employee: Simon].
```

This reads as "The employee known as Simon". The referent is a *conformity* to the *type label* in a concept. This example shows that Simon conforms to the type label 'Employee'.

A concept that appears without an individual referent has a *generic* referent. Such

---

[2] That also draws on an earlier introduction to CG in 1992 [7].

[3] The display form CG throughout this paper were produced using the *CharGer* software (http://charger.sourceforge.net/), one the many useful CG software tools that are catalogued at www.conceptualgraphs.org.

[4] There are other kinds of referents, such as plural ('sets' of) referents (which are rather like collections in object-orientated classes, and scalars ('measures') [8, 9]. In passing, as well as Concepts and Relations in CG there are Actors (which incidentally are not to be confused with UML Actors! [6]) Delugach is a proponent for the use of CG Actors [3].

'generic concepts' should be denoted as `[Type_Label: *]`. Writing `[Type_Label]` is merely a convenient shorthand.

Generic concepts may take up an individual referent. A unique identifier can be used to make a concept distinct. Thereby the generic concept `[Funding_Request]` might become `[Funding_Request: #1234]` with respect to `[Employee: Simon]`. This would yield:

```
[Funding_Request: #1234] -> (initiator) -> [Employee: Simon].
```

If there are two or more employees with the name Simon, we would then need to make them distinct from one another e.g. `[Employee: Simon#122014]`.

3 The Type Hierarchy

In CG, type labels belong to a type hierarchy. Thereby:

```
Manager < Employee.
```
("A manager is an employee")

This means Manager is a more specialised type of the type Employee i.e. Manager is a *subtype* of Employee. Alternatively, this can be stated as Employee is a *supertype* of Manager. (Subtypes and supertypes are analogous to subclasses and superclasses in object-orientation, thus a subtype inherits the characteristics of its supertypes.) Similarly, the remainder of the hierarchy may be:

```
Employee < Person.
Person < Animal.
Animal < Entity.
Entity < T.
Funding-Request < Request.
Request < Act.
Act < Event. Act
< T.
```

Sowa provides a conceptual catalog that includes a representative set of hierarchical concepts, as well as relations [9]. It also shows the context in which a type label should be used. For example an Act is an Event with an Animate agent:

```
[Act] -> (agent) -> [Animate].
```

The type denoted as 'T' means the universal supertype. It has no supertypes and is therefore the most general type.

A subtype can have more than one immediate supertype. For example, consider the concept `[Animal]`, which has a more detailed set of supertypes than indicated above. This concept has a type label Animal which may be defined as [9]:

```
Animal < Animate, Mobile_Entity, Physical_Object, ¬Machine.
Animate < Entity.
Mobile_Entity < Entity.
Physical_Object < Entity.
```

An animal therefore inherits the characteristics of being animate, a mobile-entity, and a physical object, but it is not a machine (¬ means 'not').

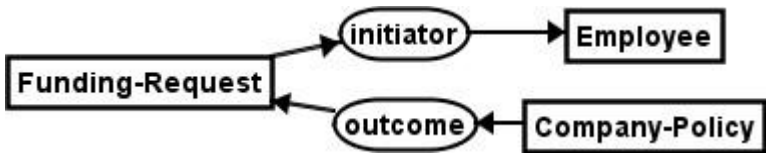
4 Projection

Projection in CG extends the notion of projection in SQL in database systems [1, 8, 9]. It is used in conjunction with specialised graphs. A CG becomes more specialised when one or more of the following happens:

a) more concepts, types and relations are added to it to narrow the scope further, e.g. an employee initiates a funding request that is the outcome of a company policy (as opposed to the earlier where it is only shown that an employee initiates a funding request) i.e.:

```
[Funding_Request] –
 (initiator) -> [Employee]
 (outcome) <– [Company_Policy].
```

This larger graph illustrates the use of hyphen '–', which allows the relations of a concept (being Funding-Request in this case) to be listed on subsequent lines. In the visual display form of CG this is not needed as the following equivalent CG shows:



b) it acquires non-generic referents, e.g. `[Employee]` becomes `[Employee: Simon]`, or
c) subtypes are substituted for (super)types e.g. replacing `[Employee]` with `[Manager]`

The following example illustrates a combination of these:

```
[Funding-request] –
 (initiator) -> [Manager: Susan]
 (outcome) <– [Company-Policy: #CP76321].
```

Therefore, each specialised graph may have a more general graph from which it was derived. Likewise, a general graph can have a number of specialised variants. Thus, *projection* is where we take a graph and try to see if another graph is a generalisation of it. It there is such a fit we have determined that the graph is indeed a specialised variant, and that the other graph is indeed a generalisation of it. We can then state that the general graph 'projects' into the specialised one.

Projection plays an important role in the inference for new knowledge with conceptual graphs. If a graph projects into another, then a particular pattern may have been identified. Projection may result in a new graph being asserted. *Inference*, the

operation that describes these assertions, is discussed later. Projection also features in the *combining* of graphs to form larger graphs. This subject is discussed first and will serve to illustrate projection and help us to understand inference better.

5 Combining Graphs

The joining of graphs facilitates inference because more projections can be made into bigger graphs. *Maximal join*, which extends the notion of join in SQL in database systems [1, 8, 9], defines the optimal method by which graphs are joined. Maximal join occurs when graphs are joined on the most common, or *maximally extended*, projection. This is illustrated by the P-H Co. case study in **Fig. 1** as follows:

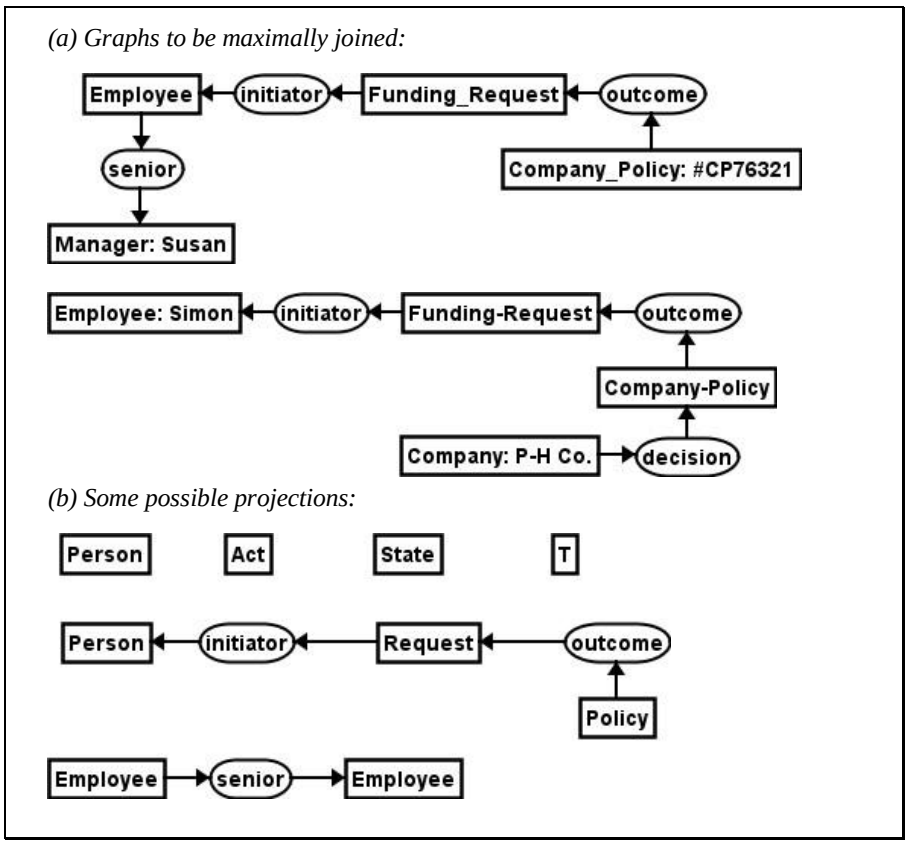


Fig. 1. Maximal join

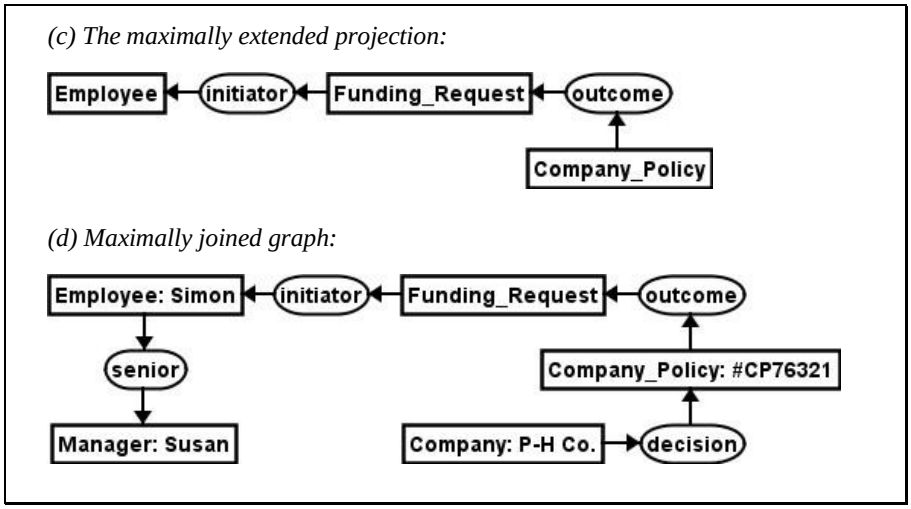


Fig. 1. (*continued*)

For this figure, the type hierarchy elements are (in addition to those given earlier):

```
State < T.
Policy < State.
Company-Policy < Policy.
Company < Entity.
Manager < Employee.
```

Having stated all this, such joining of graphs may lead to invalid results. This obstacle arises because Sowa treats the generic referent as theoretically equivalent to the *existential quantifier*, '∃', in predicate logic[5]. ∃ means a conditional "there exists an item such that ..... ". For instance, a graph which has `[Person]`, `[Person: *]`, or `[Person: *x]` in it means 'There exists a person (or some person) such that the other concepts and relations which make up the attached graph are valid'. However, this comparison is violated when graphs are combined because *any* item will be suitable as a referent in a generic concept provided it conforms merely to the type label in the concept. This pays no regard to the conditional statement given by any concepts and relations attached to it. `[Person]` is treated as *any* person when it should mean an *unknown* person.

This can be illustrated in **Figure 1** where it is quite possible for instance that an employee other than Simon has Susan as his manager, or that Susan is a manager who in fact works for a different company to P-H Co.! Joining should thus only occur when the referents are known to match in the graphs to be joined. It is CG's inference capability that can assist in determining this knowledge.

[5] Conceptual graphs are, in fact, an existential notation, as there is a direct mapping between conceptual graphs and first order predicate logic.

6 Inference

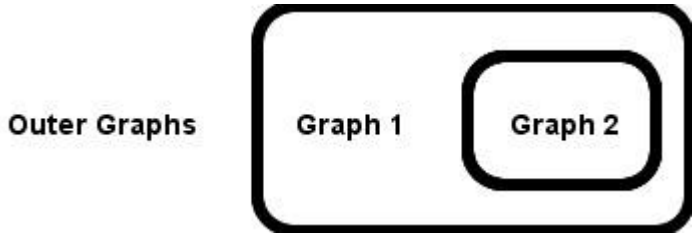
Inference in conceptual graphs theory is based upon the existential graphs logic of Charles Sanders Peirce[6]. This 'Peirce logic' is developed by Sowa in the same principled way to provide a comprehensive inference capability in conceptual graphs. Peirce logic, cited by its founder as 'the logic of the future', is described by Sowa as an enhancement of the traditional propositional and predicate logic of Peano, Russell, and Whitehead [9].

Consider the following example (where we have simply referred to graphs as 'Graph 1' and 'Graph 2' as this allows us to focus on how the inference operates; a fuller example involving actual CG will be described later):

if Graph 1 **then** Graph 2.

This may be read as "If Graph 1 can project into any outer graphs, then Graph 2 can be asserted".

In Peirce logic, 'if-then' can be rewritten as: **not** (Graph 1 **and not** Graph 2). This can be written graphically in Peirce logic as:



This visual form illustrates the contextual domination of graphs over other graphs. A graph is dominated by another graph if the dominated graph is ringed by what is known as a *negative context*[7], whereas the dominating graph is outside of that ring. Here, the outer graphs dominate Graph 1 which dominates Graph 2.

Any graph which projects into a graph which dominates it may be 'rubbed out' or *deiterated*. To assert Graph 2, Graph 1 must project into the outer graphs. Should this occur, Graph 1 can be deiterated leaving two rings around Graph 2 thus:

The term '**not**(**not** Graph 2)' equates to the term 'Graph 2' so the empty outside ring and the inside ring cancel out, or *double negate*. This frees Graph 2 out of its contexts

[6] Pronounced as 'purse'.
[7] Peirce referred to these as 'cuts' [9].

and thereby means it has been asserted as a new graph in the outer graphs. Graph 2 is thus true. This example demonstrates that a true antecedent in an 'if-then' rule means its consequent is always true. This is the general inference rule of *modus ponens* and has been demonstrated here using Peirce logic.

The linear equivalent of the above is:

```
¬[Graph 1 ¬[Graph 2]].
```

As we know, the symbol '¬' means 'not'. A '¬[....]' forms a negative context ring. Thus '¬[Graph]' means that graph is not true. The term 'not true' in this sense equates to that graph being *false*. Therefore, it is also possible by the appropriate use of negative contexts and nested negative contexts to build a knowledge base consisting of both true graphs, false graphs, and various inferences of those graphs. For the sake of clarity, this discourse will substitute '(....)' in place of '¬[....]' to denote negative contexts written in the linear form.

Using this preference, the linear equivalent of the above example is:

```
(Graph 1 (Graph 2)).
```

Consider the next example: Graph 1 **and** Graph 2. This is merely a case of adding Graph 1 and Graph 2 to the outer graphs because they both are true i.e.:

```
Graph 1 Graph 2.
```

Say, however, the example was:
if (Graph 1 **and** Graph 2) **then** Graph 3.

In Peirce logic form this would be:

```
(Graph 1 Graph 2 (Graph 3)).
```

If Graph 1 and Graph 2 existed in the outer graphs, then they can be deiterated and Graph 3 double negated thereby asserting it as a new graph. Now say that the outer graphs happened to include the graph `(Graph 3)` instead – i.e. Graph 3 is false. Regarding the above rule, `(Graph 3)` can be deiterated from it leaving:

```
(Graph 1 Graph 2).
```

This shows that because Graph 3 is false then *both* Graph 1 and Graph 2 are false. It is still possible for *either* Graph 1 *or* Graph 2 to be in the knowledge base *but not both*. If they were or some derivative that would state they were, this would show there is an inconsistency in that knowledge base. Return to the first 'if Graph 1 then Graph 2' example:

```
(Graph 1 (Graph 2)).
```

Should `(Graph 2)` be in the outer graphs, then `(Graph 1)` would be asserted. This demonstrates another general inference rule of *modus tollens*, or that if the consequent ('then' part) of an 'if-then' rule is false then so is its antecedent ('if' part). The illustration also shows that if the antecedent is false, then the consequent cannot be determined from it. If `(Graph 1)` was in the knowledge base to begin with, there is no possible way to assert either `Graph 2` or `(Graph 2)` from `(Graph 1)` alone.

It should be noted that if a graph is false then so will be any of its specialisations. If "the employee Simon" is false, then "the employee Simon who works for P-H Co." (or indeed any employer) must also be false.

As a final example, consider: Graph 1 **or** Graph 2. Logically, 'or' can be rewritten as: **not** (**not** (Graph 1) **and not** (Graph 2)). This maps to the Peirce logic form:

```
((Graph 1) (Graph 2)).
```

By the above discussed Peirce logic operations, if either Graph 1 or Graph 2 was false then Graph 2 or Graph 1 would be true respectively. It also shows that 'or' is the same as 'if not … then'. Thus '**if not** Graph 1 **then** Graph 2' is the same as 'Graph 1 **or** Graph 2', as is '**if not** Graph 2 **then** Graph 1'.

An example using actual CG is given by **Fig. 2**:

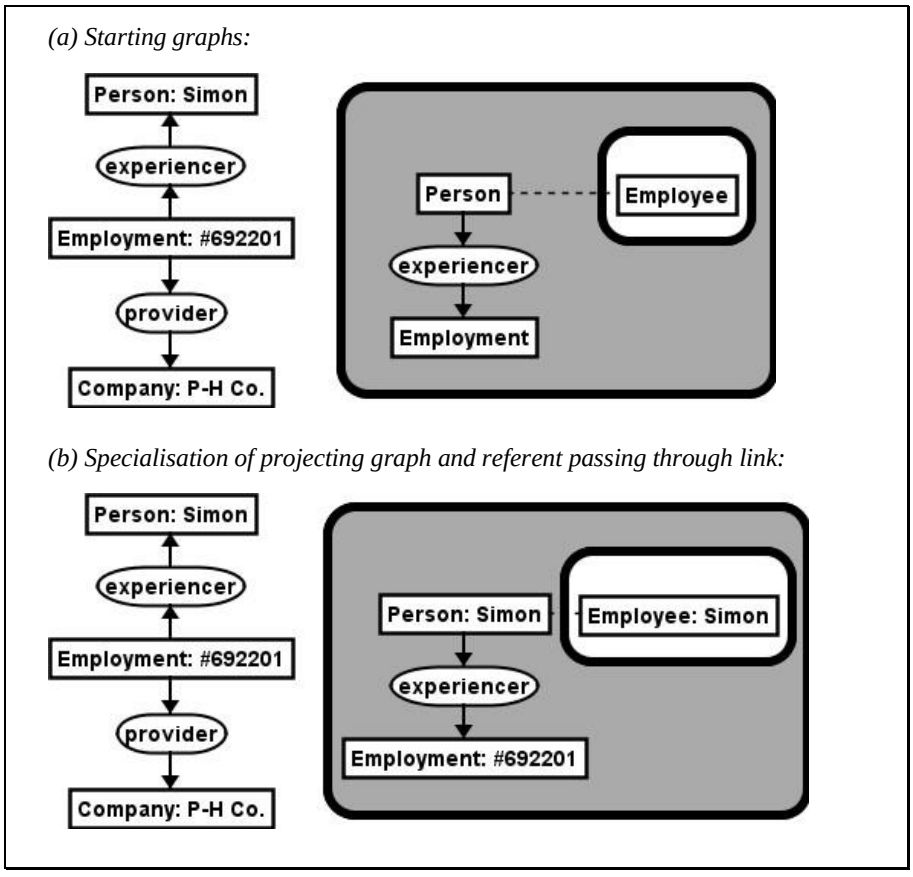


Fig. 2. An actual conceptual graphs inference

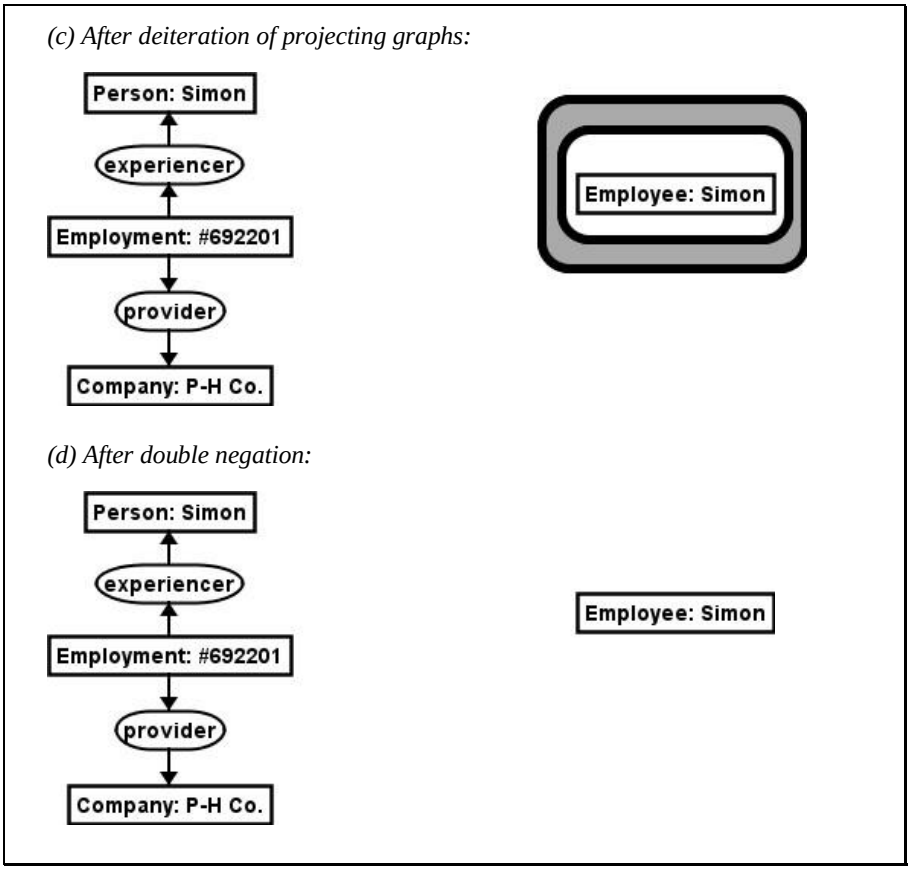


Fig. 2. (*continued*)

This figure continues the P-H Co. example as before and shows that if a person is in employment that person is an employee, and that this is true for Simon. We also need to add that:

```
Employment < State.
```

Fig. 2 shows that before any projecting graph can be deiterated, it must first be specialised, and any referents passed onto all other *co-referent* concepts. In the display form the co-referent can be shown as a dotted link, which can be seen in this figure.

Whilst **Fig. 2** demonstrates modus ponens with CG, we can also demonstrate modus tollens with the following CG that shows that it is false that Simon is an employee i.e. `([Employee: Simon]):`

```
([Person:*x] <- (experiencer) <- [Employment] ([Employee:*x])).
```

Here the co-referent is shown by the alternative of matching `*x` values. After specialisation:

```
([Person: Simon] <- (experiencer) <- [Employment] ([Employee: Simon])).
```

After deiteration:

```
([Person: Simon] <- (experiencer) <- [Employment]).
```

Hence as it is false that Simon is an employee, he cannot be experiencing employment!

In summary, Peirce logic shows contexts of knowledge elements visually dominating others and inference is performed by attempting to reduce those contexts. As we can also see, it also sets the conditions by which referents can be correctly instantiated thus addressing the issue identified by the earlier **Fig. 1** discussions.

7 A More Comprehensive Illustration

Whilst the foregoing examples demonstrate the elegance of CG, real-world information systems would need to handle much more complex problems. The following is a more realistic illustration of a typical problem, again referring to the PH Co. funding request scenario as the ongoing case study. Note that even this example must be necessarily simple so as to get the ideas across, but that it more fruitfully suggests how CG can be put to work in the real world.

Let us therefore refer to the following statement as our starting point:

P-H Co. Company Policy # PHCP69692*. There have been no guidelines to help P-H Co. allocate its budget for funding requests by employees for new business projects. There is a need for a guideline so that the decisions of the company are consistent, to encourage less senior members of staff to make such requests, and to evidence this rationale. P-H Co. therefore has decided on this company policy, which is that funds will be allocated in the following order (highest priority first): Junior Staff, Senior Staff, Manager, and lastly Director.*

The actual employee seniority relationships in P-H Co. are given by **Fig. 3**:

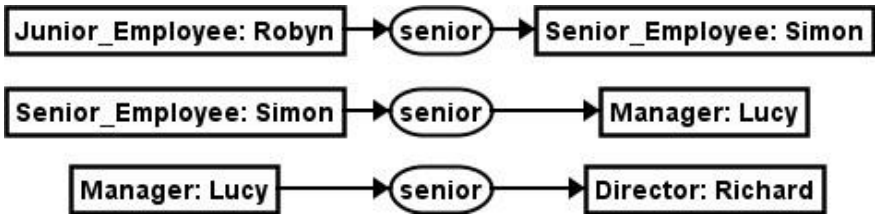


Fig. 3. The seniority relationships in P-H Co

Added to the type hierarchy are:

```
Junior_Employee < Employee.
Senior_Employee < Employee. Director
< Employee.
```

We can also describe in CG the general applicable concept of seniority, **Fig. 4**.

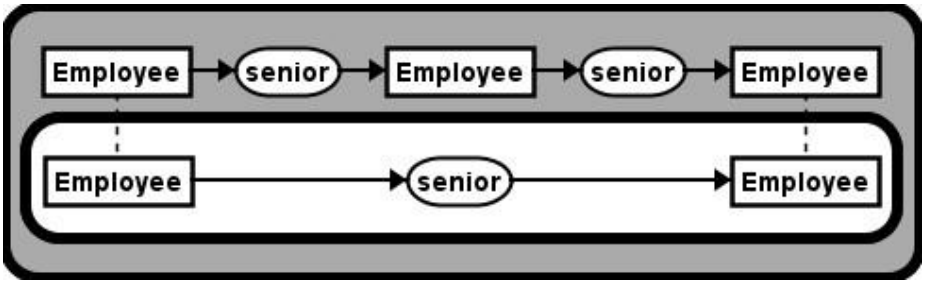


Fig. 4. The generally applicable concept of seniority

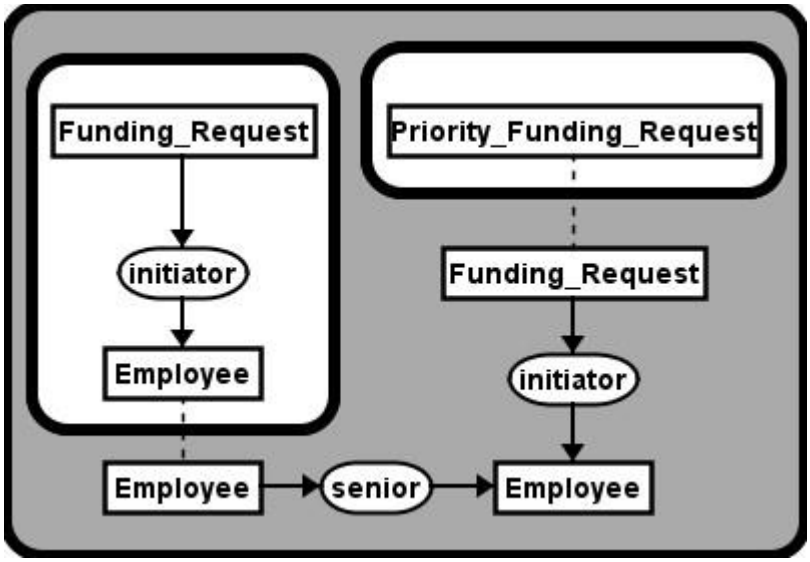


Fig. 5. Describing the funding priority element in P-H Co.'s company policy

Fig. 4 shows that seniority is transitive in nature. Thus, if we wished to establish if the Director 'Richard' was the senior of a Junior Employee 'Robyn' the above CG could determine this beginning with maximal join of the first two CG in **Fig. 3** i.e.:

```
[Junior_Staff: Robyn] -> (senior) -> [Senior_Staff: Simon] ->
(senior) -> [Manager: Lucy].
```

This is a legitimate maximal join as we had denoted that the referents match through being co-referent. The antecedent component of **Fig. 4** can project onto this CG and specialise to match it:

```
[Junior_Staff: Robyn] -> (senior) -> [Senior_Staff: Simon] ->
(senior) -> [Manager: Lucy].
```

Continuing, the lines of identity (co-referent links) would specialise the consequent part of **Fig. 4** to:

```
[Junior_Staff: Robyn] -> (senior) -> [Manager: Lucy].
```

The antecedent can then be deiterated and, after double negation, the consequent would be asserted. It can, in turn, be maximally joined with the third CG in **Fig. 3 to** give:

```
[Junior_Staff: Robyn] -> (senior) -> [Manager: Lucy] ->
(senior) -> [Director: Richard].
```

This forms a new antecedent CG in **Fig. 4** (after specialisation, deiteration then double negation) to give the asserted CG:

```
[Junior_Staff: Robyn] -> (senior) -> [Director: Richard].
```

Thus, Richard is senior to Robyn. Using this background knowledge, we can draw a CG rather like **Fig. 5** to describe the company policy. This more comprehensive CG models the priority of funding that favours less senior employees. Essentially, if a funding request is initiated by a more senior employee then that will become a priority funding request provided a less senior employee has not initiated a funding request. To appreciate the lucidity of this CG (and the previous ones), try them out with various scenarios (including modus ponens and modus tollens), following the steps of projection, maximal join, specialisation, deiteration and double negation as appropriate.

To assist, Aalborg University provide a comprehensive online course on CG (www.huminf.aau.dk/cg). For a practical implementation of Peirce logic in CG, refer to Heaton [5]; for further theoretical discussion see Sowa and Dau [2, 9].

Also **Fig. 5** might be adapted in the light of new circumstances: P-H Co's funding policy would need to be adapted if there was more than one employee of the same grade; the new CG may then need to reflect that, for example, priority is then given according to the proposal being ranked by a panel of judges. We can therefore see that this simple funding example is beginning to take on real-world dimensions.

8 Concluding Remarks

We can see that CG are variable-sized, hierarchical structures. Projection, maximal join and inference show how value is being added to data as CG capture the underlying concepts behind data, relate data to other data, and find patterns from them. By capturing the meaning behind data, CG therefore capture knowledge in a way that is more useful to people whilst bring able to be directly represented in software.

Whilst many in the CG and CS community will find little that's new in this introduction (albeit it might provide a useful refresher), it is hoped that it will encourage newcomers to engage with this exciting field and to help realise "Conceptual Structures: Knowledge Architectures for Smart Applications", the theme of ICCS 2007.

References

1. Connolly, T.M.: Database Systems: A Practical Approach to Design, Implementation, and Management, 4th edn. International Computer Science Series. Addison-Wesley, Harlow (2005)
2. Dau, F.: The Logic System of Concept Graphs with Negation. LNCS (LNAI), vol. 2892. Springer, Heidelberg (2003)
3. Delugach, H.: Active Knowledge Systems for the Pragmatic Web. GI P-89, 67–80 (2006)
4. Ganter, B., Stumme, G., Wille, R.: Formal Concept Analysis: Foundations and Applications. In: State of the Art Survey. LNCS, Springer, Heidelberg (2005)

5. Heaton, J.E.: Goal Driven Theorem Proving using Conceptual Graphs and Peirce Logic. Doctoral Thesis, Loughborough University (1994)
6. Mattingly, L., Rao, H.: Writing Effective use Cases and Introducing Collaboration Cases. The Journal of Object-Oriented Programming 11, 77–87 (1998)
7. Polovina, S., Heaton, J.: An Introduction to Conceptual Graphs. AI Expert 7, 36–43 (1992)
8. Sowa, J.F.: Knowledge Representation: Logical, Philosophical and Computational Foundations. Brooks-Cole (2000)
9. Sowa, J.F.: Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, London, UK (1984)
10. Wolff, K.E., Pfeiffer, H.D., Delugach, H.S. (eds.): ICCS 2004. LNCS (LNAI), vol. 3127. Springer, Heidelberg (2004)