# IS-A Object PART-OF Knowledge Representation? (part two)

**Tim Menzies**
Artificial Intelligence Lab
School of Computer Science and Engineering,
University of New South Wales
P.O. Box 1, Kensington, NSW, Australia, 2033
email: timm@spectrum.cs.unsw.oz.au

A paper for the fourth International Tools Pacific Conference,
TOOLS 4, Sydney, Australia, December, 1991.

**ABSTRACT:** The utility of objects as a knowledge representation (KR) schema is discussed. Objects will be found to be useful as a software engineering tool, and as an tool for explicitly representing control knowledge. However, commonly used object-oriented (OO) systems (e.g. C++, Smalltalk, and Eiffel) have will be found to be incomplete KR tools.

**KEYWORDS:** Objects, frames, logic, knowledge representation, expert systems.

## 1. INTRODUCTION

In recent years, there has been much interest in developing object-oriented expert systems (ES). Many ES practioners have realised that objects are very close to the knowledge representation (KR) concept of a *frame* [Minsky 75]. Many features of the frame model are present in the object-oriented (OO) model[1]. This realisation suggests that commercial OO systems such as C++, Smalltalk, and Eiffel are suitable tools for knowledge representation (KR). This paper will argue that this is not the case and that while systems such as these can be used as KR tools, they require modification.

This paper continues a discussion that began in [Menzies 90a & Menzies 90b]. At issue is the pragmatic utility of OO techniques for commercial knowledge engineering. The feedback received by the author from the previous papers seemed to confuse three issues: OO as a software engineering tool; OO as a method of explicitly representing procedural knowledge; and OO as a KR tool. This paper will argue each of these issues separately.

## 2. SOFTWARE ENGINEERING

This section discusses the use of objects for ES from a software engineering (SE) prospective.

---

1. Frame slots can be modelled as instance's private data. Instantiating a generic frame is akin to creating an instance. Frame demons can be implemented as methods (as can slot-fillers). When searching for knowledge that matches the current situation, the frame system can take a guess, instantiate the guessed frame (i.e. create a new instance), and then query the instantiation to check its validity. If in valid, the frame's hierarchy (super-classes) can be explored looking for a frame whose instantiated version is valid.

Consider the standard SE cycle:

```
specification -> analysis -> code -> test
```

Expert systems can be characterised as *executable specifications*. Ideally, the knowledge base (KB) author expresses their requirements in a form that is instantly executable. KB authors can then get rapid feedback on the completeness of their requirements. This feedback can prompt a rapid evolution in the specification.

```
start -> change spec -> test -> goto start
```

The software engineering challenge for ES is to support the execution of a specification that is expected to change significantly in short periods of time. In the context of supporting an evolving specification, many of the software engineering benefits of OO have a special significance for expert systems.

According to [Meyer 88], objects are best viewed as a *repository of services* that can be offered to some client. No assumptions should be made about the order in which the public interface methods of a well-designed object are called. Such assumptions complicate maintenance when the calling order of the functions may be re-arranged due to a change in the specification.

Objects can usefully *hide information*. The KB author can be shielded from procedural complexity via object interfaces. In such a "cushioned" environment, a non-technical domain expert could develop a KB. For example, in a hybrid rule-based/ object-based ES, a rule could read:

```
rule300
if      the creditRating of an applicant
                is medium and
        the age of an applicant
                is below 25
then    reject(an applicant)
```

**Figure 1.** A rule using procedures defined in an object system.

The function *creditRating* could protect the KB author from numerous implementation details. For example, *creditRating* could execute a complicated join across a database to extract the pre-computed credit rating. Alternatively, it could access some mathematical credit scoring system that awards numeric points to the applicant's application. Similarly, the *age* function could compute the applicant's age by computing the difference between the date of birth stored in the applicant's instance and the current year.

The ability to sub-divide a large problem into smaller, more manageable "chunks" is a useful design tool. One of the draw-backs to most rule-based expert systems is the global nature of all knowledge. Facts and rules live in the one global data-space. The assertion of a fact can have unforeseen consequences as it affects one of the rules in the global space in some odd way. Many OO ES use objects to divide up the reasoning. For example, [Aikins 83]'s *CENTAUR* system associates rules with objects. The rules are fired as a side-effect of processing that object. Further, the use of rule group objects (called *contexts*) can explicitly represent logical groupings that may be only implicit in a pure rule-based system. So, instead of:

```
Rule 1: if A and B and C then X
Rule 2: if A and B and D then Y
Rule 3: if A and B and E then Z
```

**Figure 2.** Implicit context in a global rule-space.

rule knowledge can be represented as:

```
Context: A and B are true

Rule 1: if C then X
Rule 2: if D then Y
Rule 3: if E then Z
```

**Figure 3.** Explicit context in a divided rule-space.

This re-expression is more than just a mere syntactic edit on the original rules. Using contexts, knowledge relating to one topic can be grouped together in one place. This makes subsequent edits easier.

Finally, objects can institutionalise the concept of *dialogue independence* : i.e. a clear separation of a program's computational component from its interface. Such a

separation permits the separate subsequent modification of a program's interface or functionality without having to re-write the entire system [Hartson 89]. OO interface toolkits can directly implement dialogue independence by assigning separate objects to a program's *model* and its *view*. Certain OO interface toolkits extend dialogue independence even further than a two-way split. Smalltalk assigns a model, view, and a *controller* object to each pane of each window. The controller object processes user keystrokes and mouse movements. Objective-C's interface kit adds a fourth object called a *transparent controller* that manages the background pane of a window (which the user sees as the top-bar of the window). See [Urlocker 89] and [Knolle 89] for more details. [Menzies 90a] argues for an extension to the basic MVC-triad and describes five new classes for ES with an extensive interface component.

## 3. INFERENCE CONTROL

This section discusses OO KR from a inferencing control perspective.

[Aikins 83 & Lenat 83] argue that OO KR can be used to explicitly and usefully model control strategies. Aikins notes that the following "rule" from the [Kunz 78] *PUFF* system is less concerned with the knowledge of a domain expert than the ordering of the reasoning:

```
If: 1) An attempt has been made to deduce
       the degree of disease X
    2) An attempt has been made to deduce
       the subtype of disease X
    3) An attempt has been made to deduce
       the findings about the diagnosis
       of disease X
Then:
   It is definite (1.0) that there is an
   interpretation of potential disease X
```

**Figure 4.** Implicit control in a rule.

This rule's purpose is to invoke other rules that finds, then refines, evidence for some disease. Re-ordering the "logic" of the premises would cause an inappropriate search for a refinement of a diagnosis before checking that the diagnosis is in fact relevant. *CENTAUR* stores such ordering information explicitly in slot contents.

Lenat's RLL system stores its "rules" as instances of a rule class. For example:

```
Rule#332
Isa:    Rule
Description:
        Tell the user to hold his breath
        if the chemical is toxic.
IfWorkingOnTask:
        AscertainImminentDanger
IfPotentiallyRelevant:
        (chemical toxicity is High ?)
IfTrulyRelevant:
        (chemical Location is
                   (Nearby user ?))
ThenTellUser:
        "Do not breathe this chemical!"
ThenAddToAgenda:
        (SummonAbulances WarnOthers)
Priority:
        High
Worth:
        900
AvgRunningTime:
        .1 seconds
FrequencyOfUse:
        considered 985 times, used 4 times
Generalizations:
        (Rule#899 Rule#45)
Specializations:
        (Rule#336)
Justification:
        Breathe&DieScenario
Author:
        Johnson
CreationDate:
        17:30 on 9-July-81
```

**Figure 5.** Frames slots relating to a rule.

This OO representation of the rule makes the control explicit and permits customisation of the inferencing.

- Rather than exhaustively searching all the rules, the inference engine could take a quick "peek" at this rule to see if it might be relevant (i.e. execute the *IfPotentiallyRelevant* slot).
- If more than one rule might be relevant, then the inference engine could choose between the possible rules in a variety of ways. The rule that runs fastest (i.e. has the smallest *AvgRunningTime* figure) could be tried first. Alternatively, the rule that works most often (i.e. has the highest

"used" figure in *FrequencyOfUse*) could be fired.

- The *ThenTellUser* slot could be disabled so that the system can find all its conclusions without sending strings to the screen.
- This representation also supports certain meta-queries that simplify knowledge maintenance. For example, if rules are expressed as the above, then it is possible to find all the rules which have (e.g.) never fired.

Explicit knowledge of the structure of the procedural reasoning allows a system to reason about its own reasoning. Aikins presents an example in which the same problem is approached using three different strategies:

- *Confirmation:* attempting to confirm the most likely frame.
- *Elimination:* attempting to eliminate the least likely frame.
- *Fixed-order:* exploring all the frames in a pre-set order.

Each strategy was then assessed according to computational efficiency and user acceptability[2]. Since the strategies are explicit in slot contents, it is possible for the system to automatically learn which strategy is best to apply and modify its own future reasoning to take advantage of this learnt improvement (i.e. re-write the *strategy-slot* contents).

In a more experimental approach, [Lenat 84] describes *EURISKO*, which is a system that made extensive use of its explicit knowledge of its procedural structure to learn new heuristics. As an example, *EURISKO* applied this heuristic to the rule system shown in Figure 5 and learnt that:

> *It's usually okay to mutate a heuristic by changing and AND to an OR in its* ifPotentiallyRelevant *slot, but usually not in its* IfTrulyRelevant *slot.*

Lenat's work is very experimental and is not currently commercially viable. However, the general lesson here is that if control of inferencing is required for an application, then objects are an excellent implementation tool.

2. For example, which strategy pesters the user with the *least* number of questions.

# 4. KNOWLEDGE REPRESENTATION

This section discusses OO ES from a knowledge representation perspective.

## 4.1 The Meaning of "IS-A"

Many commentators have remarked on the confusing semantics of *is-a*. [LaLonde 91] argue that novice OO-analysts often confuse three distinct meanings of *is-a*: subclassing, subtyping, and specialization. Having presented their analysis, they then observe that it is incomplete:

> *Although we have clearly shown that is-a and subtyping subclassing are different, our definition of the is-a relationship is rather imprecise. It would be a worthwhile research endeavor to develop a more precise definition.* [LaLonde 91]

The artificial intelligence community has been struggling to precisely define *is-a* for over a decade. Rigorous studies of *is-a* can be found in [Brachman 83, Brachman 85b, Touretzky 86, & Horty 90] (as well as the [Etherington 83 & Hayes 79] papers mentioned below). Brachman's *KL-ONE* inference engine used its object hierarchies for its classification algorithm. *KL-ONE* matches a specified concept and studies the known frames looking for known concepts that subsume the new concept. If some are found, the new concept is added in the appropriate place. Otherwise, it attempts to add the new concept as a subsumption of existing concepts. *KL-ONE*'s reliance on its taxonomy for its inferencing forced Brachman to precisely

define *is-a*. [Brachman 83] is a catalogue of the various usages of *is-a*. The catalogue includes:

- set membership;
- predicates that act on a individual (e.g. Clyde is a camel means *camel(cylde))*;
- sharing of typical properties;
- superset/ subset;
- specialisation/ generalisation;
- kind (which Brachman argues is subtly different from specialisation);
- conceptual containment (e.g. a triangle is a polygon with three sides);
- abstraction (which Brachman argues is not the same as generalisation)

After presenting the catalogue, Brachman makes the following surprising observation:

> *One important observation to be made about our analysis of the* is-a *link is that inheritance of properties has played no part in our understanding.. the useful (semantic properties of* is-a*) are... not "pass this property or "block this one".* [Brachman 83]

A semantic property not catalogued above is the unique ability of inheritance hierarchies to override inherited properties. [Brachman 85b] warns that such a feature removes any representational power from the *is-a* link. If a representational system allows for the ad hoc alteration of the relationships between objects in that representation then, almost by definition, that system has removed any consistent interpretation of the relationship between objects in that system. If we state that:

> *John is-a bachelor*

but then allow *John* to override any or all of the properties of *Bachelor*, then our statement says very little that is necessarily true about the current relationship between *bachelors* and *John*. Brachman presents this argument more amusingly as follows:

> *Q- What's big and grey, has a trunk, and lives in trees?*
> *A- An elephant- I lied about the trees.*

Brachman's implementation of *is-a* (as seen in *KL-ONE*) is based on strict subsumption. An object is in its "right" place in the object hierarchy if it is below all descriptions that subsume it and if it is above all the descriptions that it subsumes.

Touretzky and Horty are not so strict as Brachman. They permit property overrides and then discuss how to reason consistently from the resulting network. For example, consider the "Nixon Diamond".

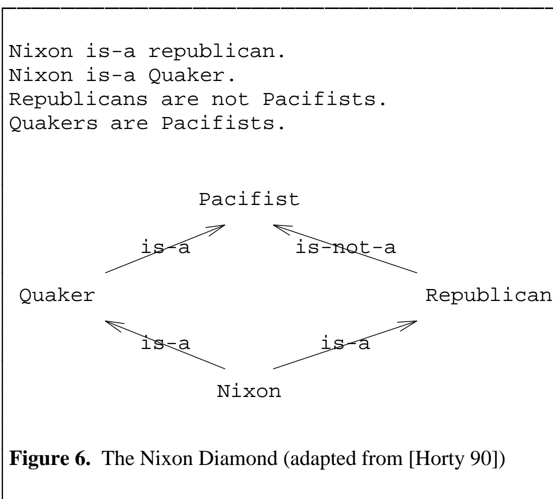


**Figure 6.** The Nixon Diamond (adapted from [Horty 90])

This network has no single interpretation. In one view, Nixon is a pacifist and in another (conflicting) view, Nixon is not a pacifist. The [Etherington 83] system forks one *extension* for each interpretation. Etherington & Reiter define a "correct" inference engine as one whose conclusions all share the same extension. [Horty 90] defines an inference engine[3] that makes reasonable conclusions from networks like the above (e.g. Nixon is a pacifist and Nixon is not a Pacifist) and avoids irrelevant conclusions (e.g. Nixon is a Democrat). Subsequent work identifies

problems with this inference procedure. Their general conclusion (as presented in [Touretzky 91]) is that inheritance systems should avoid the ability to override, then re-instate properties as this is semantically undesirable. In a limited sense, this is agreement with Brachman's earlier conclusion. Brachman cautions against overrides in a KR system while Touretzky and Horty caution against the re-instatement of overridden properties.

### 4.2 Frames <= Logic

[Hayes 79] re-expresses the frame model in terms of first-order predicate logic. For example:

```
% frame model of John and his dog
John Smith is-a person.
John Smith.pet = Fido.
Fido is-a dog.

% Equivalent logical model
IsPerson(J.S.)
& name(J.S., "John Smith")
& pet(J.S., Fido)
& Isdog(Fido)
```

**Figure 7.** Equivalence between frame/logical model.

Hayes finds that the entire frame model can be re-written in terms of logic:

> *Most of 'frames' is just a new syntax for parts of first-order logic.* [Hayes 79]

In an effort to test this conclusion, I built an OO system on-top of a logic programming language. At issue was how much extra architecture was required to convert a relational system into an object system. If none was required, then we could deduce that objects were merely a re-expression of relational calculus. The implemented system (called *OPUN*[4]) translates its commands into the underlying implementation language (in this case, Prolog). This translation process could be batched into a compilation process that removed the runtime overheads associated with converting the OO system into the underlying implementation language (i.e. in the manner of C++).

*OPUN* used a technique called *data dictionary concatenation* to drive the translation process. The statements:

```
object isa nothing
       with slots "id".
o1 isa new object.

person isa object with
       with slots "name" and "dob".
p1 isa new person
       with name =  tim and
            dob = 1960.

worker isa person
       with slots "job".
w1 isa new worker
       with name = jane and
       dob = 1959 and
       job = programmer.
```

**Figure 8.** Statements about objects.

could be interpreted as specifying three relations (which are presented below in a Prolog syntax). Note that the knowledge of the structure of the super-classes cascaded down and is appended to knowledge about sub-classes.

```
% data dictionaries (= class definitions)
dd(object,[id]).
dd(person,[id,name,dob]).
dd(worker,[id,name,dob,job])

% tuples (= instances)
object(o1).
person(p1,tim,1960).
worker(w1,jane,1959,programmer).
```

**Figure 9.** Statements about a database (same as above).

*OPUN* supported methods. For example, the following is the definition of a *person*'s age:

3. It is interesting to note that their inference procedure is not consistent with a view of inheritance as a top-down "property-flow" of features from super-classes to sub-classes (i.e. the standard inheritance mechanism as seen in systems like C++). Rather, the processing of exception networks requires a runtime bottom-up search for properties from sub-class to super-class.

4. ***O****ptimised* ***O****ptional* ***O****bject-****O****riented* ***P****ortable* ***P****rolog* ***P****rogramming* ***U****sing* ***U****niversal relatio****N****s.*

```
?-      method age
        for     person = SELF
        args    AGE
        code    (currentYear(Y),
                SELF says dob = DOB,
                AGE is Y - DOB).
```

**Figure 10.** Defining a *person*'s age.

Prolog's meta-level predicates could then be used to build the translation system.

```
% a query in the OPUN syntax
?-      select worker and
                wage = WAGE and
                age = AGE.

% the internally generated goal:
?- worker(ID, NAME, DOB, JOB),
   salary(JOB, WAGE, _),
   currentYear(THIS_YEAR),
   worker(ID, NAME, DOB, JOB) says
        dob = BIRTH_YEAR,
   AGE is THIS_YEAR - BIRTH_YEAR.
```

**Figure 11.** Query translation in *OPUN*.

One of the benefits of implementing *OPUN* in Prolog is that we get a query language for free. This feature solved one of the problems of object-oriented databases (and by extension, knowledge bases). In his critque of OODBMS, [Date 90] observes that many OODBMS only allow the processing of objects an instance at a time. This is inadequate since users typically pose queries for a set of data at a time. *OPUN* does not suffer from this defect since its query will automatically search all the assertions in a relation as well as assist in joins across multiple relations.

This OO extension to Prolog was trivial, once the query langauge primitives were available. The code for the query system comprised some 2400 lines of Prolog, 1300 of which were a general library of Prolog code. The OO extensions were another 300 lines, most of which were to do with a user-friendly object specification shell. The actual core of the inheritance mechanism was less than 20 lines. This experiment hence did not refute the [Hayes 79] conclusion.

### 4.3 Frames Need Logic

Ignoring the relationship between frames and logic can be detrimental to the inference capability of a ES. [Etherington 83] re-expresses inheritance networks that support cancellation of properties (i.e. subclasses can override inherited properties) in terms of Reiter's *default logic*. The subsequent formal analysis demonstrates certain limitations with the inferencing capabilities of such net (for example, consistent inferences can not be drawn via a one-pass parallel inferencing algorithm under purely local control). Such limitations would not have been realised unless the informal semantics of the inheritance network had been expressed formally.

Pragmatic considerations have forced the developers of certain large OO KRs to augment their object system with a logic system. [Bracham 85a]'s *KL-ONE* is a general frame-based knowledge representational tool. Lenat's *CYC* is a ten-year project to code up all the tacit knowledge required to understand 1000 one paragraph encyclopedia entries [Lenat 86, Lenat 90a, Lenat 90b]. Both systems began as essentially frame-based systems but, over time, an extra declarative layer evolved on top of the original system. Brachman recognized the need for a *assertion* language (the declarative layer) and a separate *description* langauge (the object system). Brachman argues that assertion without description is just as useless as the other way around. The sentence

*Elephants are gray.*

is a combination of two essentially different kinds of knowledge: assertions about something (in this case, the symbol "elephants") and the somethings themselves. In order to define a sentence, we need not only knowledge about the logic of the sentence, but we also require background knowledge about the terms in the sentence. *KRYPTON* has a frame-style language for for forming terms and a first-order predicate langauge for forming sentences.

The *CYC* project reached a similar conclusion. After the frame system was instantiated, *CYC* still required additional machinery to support statements like:

> *"Bill is either a terrific fisherman or a terrific liar."*
> *"Siblings almost never has the same name."*

Hence, the *CYC* implementation language (called *CycL*) includes a constraint language. Lenat argues that the slots in his frames are akin to predicates and that since *CycL* supports variable slot names, that the constraint language is essentially second-order predicate calculus. Unlike Brachman, however, Lenat does not believe in two essentially different types of knowledge and asserts that the constraint language can express everything the frame-language can express. However, he cites software engineering reasons for keeping the frame language such as simplicity and extendability of the implementation as well as speed of deduction [Lenat 90b]. Speed is a non-trivial issue in the *Cyc* system. Lenat estimates that the system will is currently 0.1 percent finished. It currently comprises 50,000 "units" (read frames or objects). Based on these figures, we can guess-timate that the completed *Cyc* will have to reason over a search space of some five hundred million frames.

## 5. OO AND ES

If we wish to benefit from the SE advantages of OO, or if we wish to explicitly represent procedural knowledge, then we should use OO to implement ES. [Menzies 90a] discusses class libraries that are useful for OO ES development. Can we augment OO languages to avoid the KR problems mentioned above?

Firstly, it seems universally agreed that the KR should not include networks of inheriting objects which can override and re-instate inherited properties.

Secondly, we have seen that it is useful to add a declarative level to the object system (*CycL*'s constraint language, *KL-ONE*'s assertion language, and the instance query language of *OPUN*). Indeed, we have also seen that an OO KR can be replaced with by a logical formalism. How should we implement the declarative level?

- The *OPUN* experience suggests that, pragmatically speaking, it is best to implement the logic on top of the objects and not the other way around. The *OPUN* system handled queries elegantly, but updates were problematic. Queries could trigger updates as side-effects. This lead to significant problems with non-static clause lists. *OPUN*'s methods had to take great care that they were accessing the current instance variables and not some older version that have since been updated. *OPUN* would have benefitted from being implemented in a language that supported instance identity as actual pointers.
- Brachman warns us against abusing the muddled semantics of the *is-a* link. He advises that the various logical components of inheritance should be teased apart and offerred as explicitly separate services. Such a separation can be implemented as a *node/link* network.

### 5.1 Node/Link Networks

Consider the object representation required for the following three equations:

```
a + b + c = 15.
b/ h = 5.
c = 9.
```

**Figure 12.** Three equations.

At a level of abstraction, these equations are a discussion of the inter-relation of the variables *a,b,c,h* and the constants *15,9, and 15*. We can draw this relationship, as shown below.

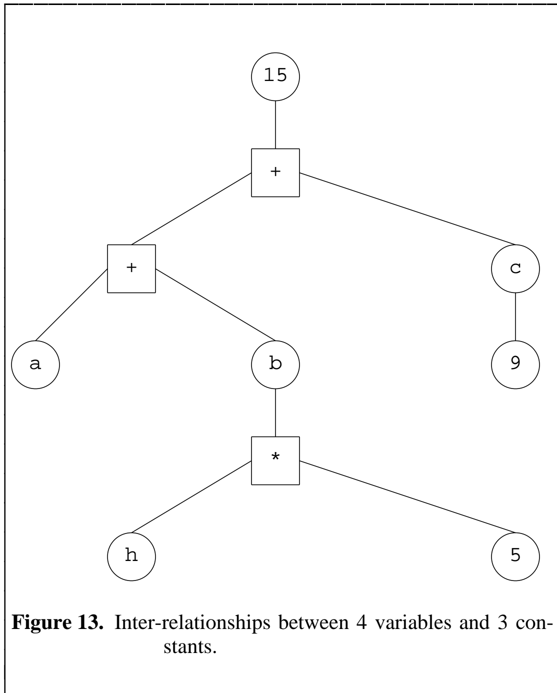


**Figure 13.** Inter-relationships between 4 variables and 3 constants.

The graph can be used to perform *constraint-based reasoning* or *direction-less arithmetic*. It can be processed to deduce values for variables that do not appear by themselves on one side of an equals sign. The node/link model for these equations is shown below.

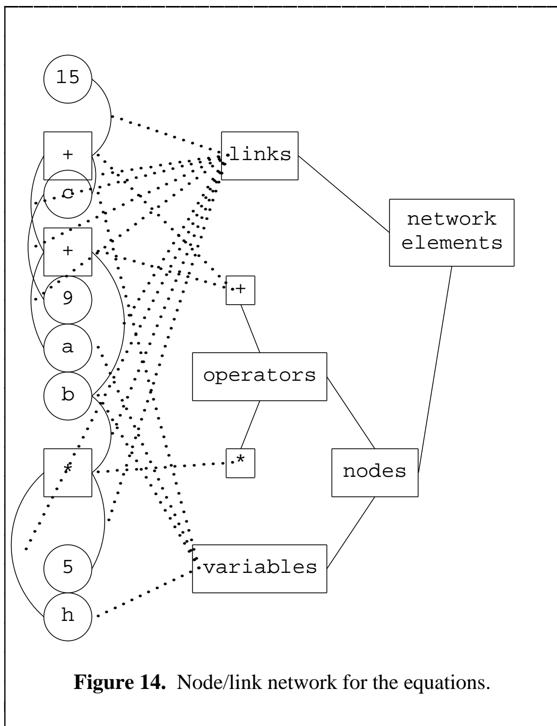


**Figure 14.** Node/link network for the equations.

This model contains some surprises. The user's specification is not stored in classes, but in instances. There is no divide object. Divide is really a re-expression of multiplication and so the system implements division as merely a fiddle on how multiplication nodes are inserted. Note also that constants are not actually objects. Due to certain esoteric implementation details, this system stored its constants as instance variables on the link objects. And that's the other surprise: link objects. Each link in the direction-less arithmetic graph is actually an object. This allows us to customise the semantics of a particular relationship into the methods of an object.

Node/Link networks permit the development of arbitrary relationships as networks of *instances*. The use of instances to store knowledge may seem counter- intuitive in an OO tool. However, consider:

- If a class is viewed as a re-usable software IC (as per the [Cox 90] model), then only concepts that migrate between applications should be made into a class. Concepts like *bank-teller* and *XYZ-Insurance Company* belong as strings stored as instance variables rather than as class names. In my view, when a software engineer creates a class, they are making a promise that its contents will be usable in another application. Domain-specific knowledge is, by definition, domain-specific. Hence it belongs in an instance.
- Many KRs could use node/link networks. Consider a binary-relationship model for a standard business application. The model might assert that *employee* and *manager* are both kinds of *people*. Such a link could be implemented using the inheritance mechanism. But what about *works-at* or *is-paid* or a host of other relations required by the model? The semantics of these none-*is-a* links still have to be implemented. A generic architecture for node/link nets could support *sub-class* as one of a range of inter-relations.
- When adding a declarative level to an object system, it is wise to avoid the

dubious semantics of the *is-a* link. There are many relationships objects can have to one another (e.g. *manager*, *employee*, etc) and it just isn't clear what *is-a* means, particularly in a system that allows the over-riding of inherited features. Node/link networks allow the design team to implement *the semantics they want* rather than the semantics they have to suffer with from the implementation tool.

## 6. CONCLUSION

After purchasing a commercial OO system, a ES developer does not have a KR system. OO languages are more a software engineering tool that a knowledge representation tool. The pragmatic software benefits of the OO approach are non-trivial and can simplify the development and maintenance of all software, including expert systems. Objects can be used to explicitly represent inference control knowledge. However, logically speaking, objects aren't logical. We have seen examples were "object logic" can be implemented by predicate calculus. Further, a rigorous analysis of a inheritance system often reveals semantic irregularities (especially when the inheritance system supports overrides).

How can Eiffel, C++, Smalltalk, *et al* be used for building KR?s

1. Experience suggest that a combination of a declarative/assertional level with a underlying definitional/object level is a useful architecture. Much domain-expert knowledge relates to multiple inter-relationships between objects. An OO-KR system needs a declarative query language for expressing these inter-relationships.

   Pragmatic considerations suggest that we should build the logic on top of the objects and not the other way around.

2. The inter-object semantics has to be customisable. Inheritance can be abused to kludge-up all manner of semantic relationships between objects. These kludges become difficult to maintain or use when their semantics are extensively explored (e.g. in large knowledge bases). In particular, overrides have to be avoided. If the developer has source-code access to the inheritance mechanism, then the semantics of inheritance *can* be altered to suite the domain. However, for most OO systems, this is not the case and the developer should use networks of instances to store the KR. Such networks have two components: *node* objects and *links* objects that implement the semantics of the links between the nodes. An editor of instance networks would be a useful generic tool for an OO ES tool.

## 7. REFERENCES


[Aikins 83] Aikins J.S. *Prototypical Knowledge for Expert Systems* in **Artificial Intelligence**, 20 (1983) pp163-210.

[Brachman 83] Brachman R.J. *What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks*, in **IEEE Computer**, October 1983, vol. 16, No. 10, pp66-73.

[Brachman 85a] Brachman R.J. & Schmolze J.G. *An Overview of the KL-ONE Knowledge Representation System* in **Cognitive Science** 9(2), April-June 1985, pp171-216.

[Brachman 85b] Brachman R.J. *"I Lied About the Trees", or Defaults and Definitions in Knowledge Representation*, in **The AI Magazine**, Fall 1985, pp80-93.

[Cox 90] Cox B. *There is a Silver Bullet*, in **Byte**, October 1990.

[Date 90] Date C.J. **Introduction to Database Systems**, Volume 1, 5th edition, Addison-Wesley, Reading Massachusetts, 1990.

[Etherington 83] Etherington D.W & Reiter R. *On Inheritance Hierarchies with Exceptions*, in **Proceedings AAAI-83**,

Washington D.C., 1983, pp104-108.

[Hartson 89] Hartson R. & Hicks D. *Human-Computer Interface Development: Concepts and Systems for its Management* in **ACM Computing Surveys**, Vol 21, No. 1, March 1989, pp5-92.

[Hayes 79] Hayes P.J. *The Logic of Frames*, in **Frame Conceptions and Text Understanding**, Metzing D. (ed), Berlin: Walter de Gruyter and Co., 1979.

[Horty 90] Horty J.F. , Thomason R.H. & Touretzky, D.S. *A Skeptical Theory of Inheritance in Nonmonotonic Semantic Networks* in **Artificial Intelligence**, 42 (1990), pp311-348.

[Knolle 89] Knolle N.T. *Why Object-Oriented User Interfaces are Better* in **Journal of Object-Oriented Programming**, Nov/Dec, 1989, pp63-67.

[Kunz 78] Kunz J., Fallat R., McClung D., Osborn D., Votteri B., Nii H., Aikins J., Fagan L. & Feigenbaum E. **A Physiological Rule Based System for Interpreting Pulmonary Function Test Results**, Working paper HPP-78-19, Heuristic Programming Project, Department of Computer Science, Stanford University, 1978.

[LaLonde 91] LaLonde W. & Pugh J. *Subclassing <> subtyping <> Is-a* in **Journal of Object-Oriented Programming**, January 1991, pp57-62.

[Lenat 83] Hayes-Roth F., Waterman D.A., & Lenat D.B. (eds) **Building Expert Systems**, Addison-Wesley, Reading Massachusetts, 1983, pp314-321.

[Lenat 84] Lenat D. B. & Brown J.S. *Why AM and EURISKO Appear to Work* in **Artificial Intelligence**, 23 (1984), pp269-294.

[Lenat 86] Lenat D.B., & Prakash M., and Shephard M. *CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks*, in **The AI Magazine**, 6, 1986, pp65-85.

[Lenat 90a] Lenat D.B., & Gutha R.V. *CYC: A Midterm Report* in **AI Magazine**, Fall 1990, pp32-59.

[Lenat 90b] Lenat D.B. & Guha R.V. **Building Large Knowledge-Based Systems**, Reading Mass, Addison-Wesley, 1990.

[Menzies 90a] Menzies T.J. *Beyond the MVC-Triad: Quality Assurance via Interactive Specification Editors*, in Meyer B, Potter J, Tokoro M., & Bezivin J. (eds) **Tools 3**, Proceedings of the third International Technology of Object-Oriented Languages & Systems conference, Sydney, 1990.

[Menzies 90b] Menzies T.J. *Is-a Object Part-of Knowledge Representation?*, Proceedings of AI'90, Perth, Australia, November, 1990

[Meyer 88] Meyer B. **Object-Oriented Software Construction**, Prentice-Hall, 1988.

[Minsky 75] Minsky M. *A Framework for Representing Knowledge*, in **The Psychology of Computer Vision**, Winston P. (ed) McGraw-Hill, New York, 1975.

[Touretzky 86] Touretzky J. **The Mathematics of Inheritance Systems**, Morgan Kaufmann, Los Altos, CA, 1986.

[Touretzky 91] Touretzky J. , & Thomason R.H. & Horty J.F. *A Skeptic's Menagerie: Conflictor's, Preemptors, Reinstaters, and Zombies in Nonmonotonic inheritance* in **Proceedings of the 12th International Conference on Artificial Intelligence**, IJCAI '91, Sydney, 24-30 August, 1991, pp478-483.

[Urlocker 89] Urlocker Z. *Abstracting the User Interface* in **Journal of Object-Oriented Programming**, Nov/Dec 1989, pp68-74

-- --