Object-Oriented and Logic-Based Knowledge Representation

Vladimir Alexiev[*]
Computing Science Department
University of Alberta
Edmonton, Alberta T6G 2H1[†]

April 1993

Abstract

This paper is a survey of a number of languages/systems based on both Object-Oriented and Logic Programming and designed expressly for Knowledge Representation tasks. My goal in the paper is to argue that the integration of these two paradigms (particularly the synergism that emerges from such an integration) forms a stable basis for Knowledge Representation at the symbolic level. I try to support this claim both by examples from the papers surveyed and by considerations in a more general context. Some more advanced topics concerning special-purpose non-classic logics are also discussed.

1 Introduction

Since mid-eighties the *Object-Oriented* (OO) paradigm has established itself as the most efficient (up to now) way to cope with the scale and complexity of large applications. KBS were one of the first to employ this paradigm exactly because they are typically large and complex. First in the LISP community (LOOPS, FLAVORS, CLOS) and then in industry (KEE, ART, NEXPERT-OBJECT), OOP was making its way. An important side benefit which OOP brings to KBS is that all components of a KBS—knowledge base, inference engine, user interface etc.—can be implemented uniformly using OOP.

However, putting aside its software engineering merits, OOP uses the same old imperative mode of programming which is used in FORTRAN, PASCAL and C. Since it is inadequate for most KR tasks, other devices such as rule-, constraint- and access-based programming were employed. Again the LISP community was the first to experiment with these ideas. Fortunately OOP, though it does not embody or encourage the non-imperative style, does not hamper it either: OOP seems largely orthogonal to this aspect. The AI community was familiar with something like OOP (Minsky's frames) about a decade before that time, but LOOPS and CLOS were the first to use OOP uniformly and systematically and to implement such important aspects as inheritance, message passing and polymorphism.

Another paradigm whose application in AI and KBS always has had numerous proponents and opponents is *Logic Programming* (LP). LP is seen advantageous for KR applications for the following reasons:

- LP is *declarative*: its user only has to state what is known about the problem domain and what is the goal, but does not has to state explicitly how to go from the one to the other.
- (Some of) The LP languages possess a clear and well-founded logic semantics which may have numerous applications ranging from algorithm complexity estimations to metareasoning about the properties of programs and knowledge bases.

However the opponents of LP state that LP is insufficient for AI applications because other considerations from cognitive science, psychology, systems science, causal models, elementary physics etc. etc. are more important. Although possibly true, this statement seems mis-aimed, because there is nothing in LP to prevent us from taking these considerations into account. It is true that logic is only a part of our understanding of the world, but nevertheless it is an important part.

A lot of work in the direction of using PROLOG for intelligent applications has been done under the auspices of the Fifth Generation Computer Systems (FGCS) project of Japan. It has been recognized very early in this project that PROLOG itself does not possess the needed software engineering devices to implement a large-scale KBS. In his paper "If PROLOG Is The Answer, What Is The Question?" [2], Daniel Bobrow argues that PROLOG is to be coupled with other paradigms (function-, rule-, access- and object-oriented) in order to be successful. Different paradigms are good in the representation of different things and "One should not be forced (metaphorically) to pry up nails with a screwdriver". Of course, one of the most important things when integrating multiple paradigms is to exploit fully the potential synergism by having a smooth integration of the paradigms, both in terms of data, execution control and underlying philosophies.

In addition to the work done for reshaping PROLOG to fit KR tasks, another important direction of the FGCS project is towards concurrent execution of PROLOG (for example the so-called "committed choice parallelism" of Guarded Horn Clauses). It turns out that it is very easy to model objects using concurrently executing PROLOG predicates which pass messages (requests to prove a goal) to each other through potentially unbound and partly-evaluated message queues (this means that a predicate (or object, or agent) starts fulfilling a request as soon as there is at least one message in the queue and does not wait for the queue to become completely known). After executing a request, the agent recreates itself by calling itself recursively with the rest of the message queue as argument. The effect of inheritance in OOP can be achieved by the actor passing ("delegating") the requests it cannot handle to one or more of its "proxys". There are numerous languages (both inside and outside of FGCS) based on this model (SPOOL, VULCAN, SCOOP, POLKA, A'UM), which is similar to the ACTORS model of Carl Hewitt and Gul Agha. However I do not cover this model in the paper for two reasons:

- I believe that before we go into concurrent computations, we need to have a good understanding of the problems in the sequential case. We should not start tackling the problems of how to parallelize a computation before we have solved its proper problems.
- Although the forementioned is a good model of OOP, it is only a model, which does not possess the software engineering (size-taming) merits of OOP.

Two of the languages surveyed below are concurrent, but I simply disregard this aspect.

There is another very important non-imperative programming style: *Functional Programming* (and also *Equational Programming*). This particular paper talks about PROLOG and not about LISP only because of my personal preferences: I like Prolog better because I believe that Prolog is closer to its theoretical roots than Lisp is to λ-calculus. Of course, having the possibility to write equations and use functions is a most welcome convenience and it can be easily implemented e.g. like in [15]. In normal Prolog, one has to write e.g. for the definition of the factorial function

```
fact(0,1).
fact(X,Y) :- X>0, X1 is X-1, fact(X1,Y1), Y is X*Y1.
```

instead of the much more natural

```
fact(0) = 1.
fact(X) = X*fact(X-1) :- X>0.
```

I would like to state explicitly my opinion that we should not restrict ourselves to Prolog (notwithstanding that most of the surveyed languages are Prolog-based), whose fixed backtracking depth-first mode of search for solutions is clearly inappropriate for some KB applications and who uses for data structuring only uninterpreted (syntactic) terms and manipulates them (constructs terms and extracts parts of terms) using syntax-based (textual) unification. On one hand, localization of the operation of Prolog into a single object may diminish the first problem, and allowing arbitrary objects as predicate arguments with unification taking into account the classification relations between these objects may solve the second problem. On the other hand, LP is not only Prolog, and Logic in AI is not only LP (I touch some additional topics in Section 4).

There is much more work to be done in developing a formal semantics of the basic notions of object-oriented computing (mutable state, object identity, inheritance etc.) which would be most useful both for Deductive Object-Oriented Databases and for Knowledge Bases. of these, mutable state seems most difficult to express in logic terms, because classic logic does not have the concept of "memory" or "state": a logical formula always has the same meaning, independent on the context or time in which we evaluate it (the technical term for this is "referential transparency").

The rest of the paper is organized as follows: in the next section I survey a number of object-oriented and logic-based languages for KR, comparing and contrasting them. In Section 3, I describe a quite different view on the role of objects in KR due to Ronald Brachman. In Section 4 I touch upon some advanced non-classical logics which may very well be applied in KR. The last section contains a summary of the claims of the paper and a discussion about what else would be useful to have in a general-purpose KR framework.

2 Languages Employing OOP+LP for KR

In this section I briefly describe a number of languages which use both OOP and LP and are expressly designed for KR and/or have been successfully used for KR tasks. For each language, I give a flavor of its syntax and semantics (typically by a small example), compare it to some of the other languages and give an account of its uses for KR known to me. My goal in this section is not that much to delve in the technical details of each language and the differences between them (some of these differences are quite subtle), but to convince the reader that the merger of OOP+LP gives a viable platform for KR.

Most of the material is adapted from the indicated papers, but I also wrote down my considerations and observations (generally all negative opinions about a language are not those of the authors of the language but are mine).

The reader is supposed to have some familiarity with PROLOG and SMALLTALK (or some other OO language). The languages are presented in chronological order.

2.1 ESP: The Fifth Generation

ESP (Extended Self-contained PROLOG) [4, 5, 13] was not explicitly designed for KR; it is the system description language of SIMPOS (Sequential Inferential Machine Programming and Operating System), the operating system of the flagship machine of the FGCS project, the PSI (Personal Sequential Inference). It is built over the PROLOG-like machine language of PSI, KL0 (Kernel Language version 0, the current version is KL1); some of the specialized commands and builtin predicates of KL0 were specifically designed and tuned up to suit ESP. ESP is an OO language, with mutable state, object classes and hierarchical multiple-inheritance structures. It has been recognized that these features of ESP very much simplified and streamlined the implementation of SIMPOS, and could be very useful for other tasks (mainly those requiring hierarchically represented knowledge) as well.

ESP is translated into KL0, therefore ESP is PROLOG-based, as opposed to OOP-based. This means that all the OO notions in ESP are to be represented in KL0 (which is basically PROLOG). But the converse is also true in some degree, because logic theories (sets of PROLOG predicates) are mapped to objects, a notion which recurs many times in subsequent languages. Another seminal notion introduced in ESP is having the ability to send a query (request to prove a goal) to an object and the treatment of this as message send. So there are two types of queries (predicate calls) in ESP: local/primitive ones, which are executed in the context of the current object (theory) and message sends (e.g. `:open(Door)`), directed to a particular receiver (here the variable `Door` is supposed to be bound to an object which can handle the message `open`).

Objects can have (in addition to sets of predicates) *slots*, or attributes which contain values changing with time. The eventual state change inside an object in response to a message send is implemented using some LISP-like construction and modification primitives present in KL0: `cons`, `rplaca`, etc. This is not accounted for by the logic semantics of PROLOG (the author says "Slot values are constants from the LP point of view"). The author agrees that this does not have a clear logic semantics (it "falls out of pure logic"), but states that since such an ability is immensely needed in a task like an operating system, they had to make a compromise. Indeed, not only at that time, but even now, there is no widely accepted logic to model mutable state. So the author was right to mention "We didn't have time to wait for such an innovation" [5, page 293].

ESP does implement in full the notions of OOP. It has metaclasses whose logic theories (methods) are responsible for the creation of new objects. It has multiple inheritance of an "accumulating type": the full logic theory of a class consists of the clauses stated explicitly in that class, merged with all the clauses defined in its superclasses. Now this creates some problems because adding clauses to a class is monotonic (the set of true consequences (provable goals) only increases), while for some KR applications non-monotonicity is critical. For example, we may want `penguin` to be a subclass of `bird` and inherit all properties of birds, except that penguins do not fly. This can be solved easily by an "overriding-mode" inheritance (if there is some clause in the subclass for a certain predicate, its presence disables all the clauses for the same predicate in its superclasses), but the solution adopted in ESP relies on the extended "cut" predicate available in KL0:

```
class bird has
instance
  :fly(Bird).
  ...
end.
class penguin has nature bird;
instance
  :fly(Penguin) :- !, fail.
  ...
end.
```

Here the cut (!) predicate commits the execution to the clause for `fly` found in the `penguin` class (which subsequently fails, i.e. returns false) and does not let it try the clause in the class `bird`. This "deep" (multilevel) cut is also useful for exception handling mechanisms needed in an operating system.

ESP has "demon" methods ("before" and "after" methods), very similar to the ones described in Section 2.6 (or better said, those ones are similar to these here, taking into account the years of publication). Another useful facility of ESP is a powerful macro language which is written in PROLOG and takes into account the specifics of PROLOG: the expansion of a macro definition may need to be distributed before and after the place where it is used (or even in the beginning and the end of the clause body) and depends on whether it is used in the clause head or body. Using this macro facility, a convenient function sublanguage has been developed, for example one may use infix `+` in the head of a clause:

```
inc(X,X+1).
```

and get the following expansion in the body of the clause:

```
inc(X,Y) :- add(X,1,Y).
```

or use it in the body:

```
  ..., p(X+1), ...
```

and get the expansion just before the literal:

```
  ..., add(X,1,Y), p(Y), ...
```

Concerning the implementation of ESP, I believe that a more appropriate approach is to implement LP on top of OOP (or make them equally basic and integrate them), and not OOP on top of LP. The author says "Method calls are 3 to 4 times slower than calling KL0 directly" [5, page 297] and also "The current implementation of ESP does not yet have the required efficiency, especially in its execution speed" [next page]. In this particular language, the dedicated hardware (PSI) and machine language (KL0) can compensate for the inefficiency, but this implementation approach may not be generally applicable. Furthermore, it seems more natural to use OOP for implementation purposes and leave PROLOG (or better some extended LP language) the full freedom to model complex domain interrelations.

In addition to the implementation of SIMPOS, ESP has been successfully used for the implementation of a natural language parser [16] using the Definite Clause Grammar paradigm of PROLOG. Each grammatical category is abstracted as a class which makes it possible to use inheritance to describe hierarchical classifications among syntactic categories. Probably there are other successful applications of ESP to KR tasks which I am not aware of.

2.2 ORIENT84/K: Concurrency, OO and LP

ORIENT84/K [11, 12] is a language and system developed by Mario Tokoro and Yutaka Ishikawa. It is the realization of a methodology which the authors call Distributed Knowledge Object Modeling (DKOM) [18] (but as mentioned before, I will not deal with the distributed aspects of ORIENT84/K here). Authors give a justification of DKOM based on a model of human behavior and knowledge processing which, although somewhat simplistic, seems convincing. They argue that the human behavior consists roughly of four steps:

1. We perceive data through our acceptors and interpret it thus transforming it to information.
2. Then we infer conclusions using our knowledge and make decisions. We may initiate the acquisition of more data and/or hypothesize and prove.
3. Based on these decisions and using our actuators, we perform actions.
4. Finally, we monitor the result of these actions (the performance of our decisions and the inference process) and use this feedback to improve both our knowledge and our actions.

Authors point out that the action of previously developed KBS has been primarily limited to step 2, while it would be useful to model the other steps not only in a robotic application, but in any large enough application which adopts the notion of interacting specialized experts which run their own deduction processes and then act according to their "beliefs" trying in cooperation to find a solution.

Authors discuss the pros and cons of imperative vs. declarative execution mechanisms and then turn to a discussion of different modularization mechanisms. They point out that in PROLOG and in rule-based (production) systems, the unit of granularity is the single knowledge fragment (clause or rule). Having access to such a (relatively) fine-granularity entities makes it easy to manipulate the knowledge base (however figuring out what exactly is the effect of a modification is not that easy), but creates problems with the management and extensibility of the KB. Furthermore, in a typical ES such as R1 (XCON), 10% of the rules may have to be used to create, sequence and maintain execution contexts and 20% of the conditions of other (operational) rules may be used to ensure that they get activated only in the proper context (in other words, to contextualise the operational rules). Even worse, using the same form of representation for operational and for control knowledge (i.e., not observing Clancey's distinction between structural and strategic knowledge), obscures the semantics of the knowledge base and makes it less maintainable. Therefore, some other mechanisms are needed for partitioning a large KB and contextualising the knowledge.

An ORIENT84/K system is a community of cooperating Knowledge Objects (KO), each of which has a *behavior* part, a *knowledge* part and a *monitor* part. The behavior part consists of imperative methods like in SMALLTALK which provide sequencing control and support the sequential execution of tasks (although strictly speaking some of these methods may be executed in parallel with other methods in the same or different KO). Here are also methods to access and modify the knowledge part. The knowledge part is a local knowledge base of rules and facts like in PROLOG which describe the declarative properties of object's internals. The monitor part consists of demons which get activated upon certain accesses or modifications of methods or predicates (or other conditions) and may be used to implement exception mechanisms and support for special cases. The three parts provide for object-oriented, logic-based and access-oriented programming respectively. The overall language looks like an extension of SMALLTALK, and its implementation is more object-based than logic-based (although technically it is quite different from the implementation of SMALLTALK).

As can be seen from this description, knowledge is localized inside the objects. As the authors note [18, page 624] and as discussed in Section 2.5, this "makes it difficult to represent general rules or interrelations among objects", however finding the exact degree in which PROLOG should be left to operate freely among objects is quite a subtle problem.

The definition of a KO class looks much like a SMALLTALK class definition:

```
DKO subclass: #AClass
    instanceVariableNames: 'i1 i2'
    classVariableNames: 'c1 c2'
AClass methodsFor: 'instance methods'
    <method part>
AClass knowledgeFor: 'instance KB'
    <Horn clauses>
AClass monitorsFor: 'instance monitors'
    <assertions (demons)>
```

Here `DKO` is the root class of the system (and the one most usually used for subclassing), `AClass` is the newly created class. (The same three categories can be present for the metaclass of `AClass`.) It can be seen that the "only" difference fro SMALLTALK is the addition of the `knowledgeFor:` and `monitorsFor` parts.

The method part, in addition to all SMALLTALK statements, can contain also KB manipulation methods: `addKB`, `appendKB` and `deleteKB` which correspond to PROLOG `asserta`, `assertz` and `retract`; and KB access methods which interface to the knowledge part: `unify(query)` which tries the query and returns true or false, and `forEachUnify(query) do: [block]` which executes the indicated block (group of statements) for each tuple satisfying the query.

The knowledge part contains PROLOG (Horn) clauses (with somewhat different syntax) and may contain queries to other objects, as well as any SMALLTALK message sends. The authors claim that the latter "preserves all the backtrack information", but do not elaborate on this. Clauses may also use any instance variables of the object, but from the PROLOG side they look like constants (the same as in ESP). An example giving a feel of the language follows (sending a mail message to the brothers of somebody):

```
instanceVariables:
  'john tom mike andy henry robert'
methodsFor: 'sending mail'
  sendToBrothersOf: x message: m
```

```
        | y | "temporary variable"
      forEachUnify(brother(x,?y))
        "Prolog query to KB part"
        do: [:z | z sendMail: m].
knowledgeFor: 'family relationship'
  "rules"
    brother(?x,?y)  | f |
        father(?x,f), father(?y,f).
  "facts"
    father(john,henry).
    father(tom,robert).
    father(mike,robert).
    father(andy,robert).
```

The monitor part is ORIENT84/K's most innovative development (in the context of OOP+LP; demons have been used in frame systems long before that). It serves as demon (gets activated upon certain types of access), supervisor (controls the concurrent execution) and guardian (ensures that certain methods can only be called when the KO is in an appropriate state). In its demon role, the monitor part may contain methods (handlers) which get called when the appropriate condition is present: `whenPredicateAdded`, `whenPredicateAppended`, `whenPredicateDeleted` match the corresponding KB manipulation methods; `whenMethodAdded` and `whenMethodDeleted` do the same when the method part is manipulated; `whenPredicateNeeded` and `whenNoPredicate` serve as "before" and "if-failed" handlers (see Section 2.6), `whenObjectNotExist` and `whenMessageNotAccepted` are used for error handlers, `whenVariableAssociated` and `whenVariableEvaluated` sanction read and write access to instance variables respectively. In its supervisor role, the monitor part may contain statements related to concurrent execution: mutual exclusion and access control, static and dynamic priorities, interrupt processing. For the guardian role, there are no specific language constructs, because the existing ones are sufficient for most applications.

To try out the expressive power of ORIENT84/K, authors have developed a solution to the well-known Hazardous Spill Emergency Management Expert System example from the book of Hayes-Roth, Waterman and Lenat *Building Expert Systems*. They comment that it was much easier to develop the system in ORIENT84/K than in PROLOG, LISP or SMALLTALK alone: in PROLOG they have had predicate name scoping problems, in LISP they represented the domain entities (buildings, chemicals and water sources) in lists and it was hard to add new entities, in SMALLTALK it would be necessary to code the inference engine and KR facilities and they could hardly remain distributed (authors haven't actually implemented it in SMALLTALK).

Another development in ORIENT84/K is a programming environment complete with browser, editor, debugger and windowing system (presumably similar to the one of SMALLTALK). I don't know whether some large-scale KBS projects were completed using ORIENT84/K, but it seems equipped with everything necessary for the task.

The significant advantage of ORIENT84/K over ESP is that ORIENT84/K is a lot more expressive and also seems more naturally implemented (with OOP put in the basis and LP implemeented on top of it). However an already mentioned problem of ORIENT84/K (which will be elaborated upon in Section 2.5) is the confinement of knowledge in the limits of individual objects.

2.3 Object-Prolog: The Power of Uniformity

Object-Prolog[6], developed in Hungary, is a very enlightening example of the sheer power of Prolog, exposed in a very parsimonious and uniform approach.

Object-Prolog features only one kind of entities called *worlds*, which serve for instances, classes, metaclasses and whatever else one needs. Worlds are uniquely named; names don't have to be simple atoms, they may be arbitrary ground (variable-less) terms. Each world has a local set of Horn clauses (so again a world is a logic theory). Worlds can be specified statically, as in

```
world symphony9.
  is-a musical-piece.
  composed-by bethoven.
endworld.
```

or created and fleshed out with knowledge dynamically:

```
?- createWorld(symphony9),
   addKnowledge(is-a musical-piece) >> symphony9,
   addKnowledge(composed-by bethoven) >> symphony9.
```

(`deleteKnowledge` is also available). It is important to note that this world creation and KB manipulation is completely backtrackable. The `>>` operator denotes message send (query) to a world, which in the case of `addKnowledge` happens to be a request to add a clause to the local KB.

The syntax of Object-Prolog uses widely the infix notation, for example `john is-father-of mary`, not `father(john,mary)`. In the following examples, reserved words are in normal font, variable and atom names are in `typewriter` font, and syntactic categories are in *italic.*

In Object-Prolog the user is given a generalized concept of inheritance and is free to ascribe to it whatever names, concepts and properties s/he sees fit. This can accomodate the inheritance relations between instance and class (instance-of), class and superclass (is-a, kind-of) and class and metaclass. The general form of inheritance in Object-Prolog is

world1 inherits *predicate* from *world2 [:- condition].*

This says that all predicates unifiable with *predicate* which are true in *world2* will also hold in *world1* (that is, *world1* will inherit them from *world2*), provided that *condition* holds (if present). (More precisely, since the inheritance is overriding, there should be no clauses in the inheriting *world1* which unify with *predicate* or otherwise they will inhibit the clauses in the superworlds). Here all items in *italics* can be arbitrary Prolog terms, which gives enormous flexibility and power. A number of examples follow to justify that claim:

- `melon` inherits `All` from `fruit.`
  Here `All` is simply a variable which unifies with everything, it is not a special keyword.

- `X` inherits `p(f(Y))` from `Z.`
  Only clauses unifying with `p(f(Y))` will be inherited.

- `X` inherits `All` from `super*X.`
  In the form `super*X`, the star is a term constructor (syntactic connective), so `super*X` is not an atomic name but a term naming the superworld. This rule says that every world `X` will inherit everything from world named `super*X` (presumably its one and only metaclass).

- `X` inherits `All` from `Y :- X is-a Y.`
  This says that a subworld inherits everything from (any of) its superworld(s) where the inheritance is determined by the user-specified relation `is-a` which may change dynamically and even be backtracked.

OBJECT-PROLOG has some means for concurrent execution and interprocess communication (borrowed from an earlier Hungarian development, T-PROLOG). Although I generally disregarded the concurrent properties of the other languages described up to here, I will describe those of OBJECT-PROLOG for the sake of the example which follows. The main concurrency primitives are:

- createProcess *(goal, world)*
  Creates a new process working on *goal* inside *world* which will terminate as soon as it proves the goal or exhausts all possibilities.

- waitFor*(message1)*, send*(message2)*
  Blocks the process until some other process sends to the world of the waiting process some *message2* unifiable with *message1.*

- wait*(condition)*
  Blocks the process until the indicated *condition* becomes true.

- hold*(time)*
  Blocks the process for the indicated period of (model, not real) *time.*

Now the reader has the necessary information to appreciate the beauty of the example which follows (a bank robbery scenario) and the complexity we can handle given only the limited means described above:

```
world bank.
  safe-type wertheim.
  safe-type chatwood.
  safe-type millner.
endworld.
```

The bank uses different types of safes.

```
world safe.
  drill opens chatwood :- hold(30).
  drill can-open chatwood.
  oxygen opens millner :- hold(40).
  oxygen can-open millner.
  \% wertheim is very strong
endworld.
```

These different safes have different properties regarding their breakability.

```
world jim.
  takes-part-in robbery.
  robbery-partner dick.
```

```
    climb-wall :- hold(10).
  endworld.
  world dick.
    takes-part-in robbery.
    robbery-partner jim.
    climb-wall :- hold(5).
  endworld.
```

There are two partners-in-crime; one of them is more agile in climbing walls than the other.

```
  world robber.
    break-into (bank, Safe) :- climb-wall,
      safe-type Safe >> Bank, waitFor(Tool),
      Tool opens Safe >> safe.
    provide-tool-for(Safe) :- wait(nonvar(Safe)),
      Tool can-open Safe >> safe, send(Tool).
  endworld.
```

This is the typical behavior of a robber: he will either `break-into` (enter the bank by climbing the wall, choose a type of safe to break into, wait for the appropriate tools to be provided by his partner and open the safe) or assist by providing the appropriate `Tool` for the particular `Safe` (wait until the `Safe` type is determined (bound), pick the proper `Tool` and then send it).

```
  world mainWorld.
    X inherits All from robber :- X takes-part-in robbery.
    get-the-money :- One robbery-partner Another,
      createProcess (break-into(bank,Safe), One),
      createProcess (provide-tool-for(Safe), Another),
      time-condition.
  endworld.
```

This is the main world of the program. The first clause states that everybody who `takes-part-in robbery` is a `robber`. The second clause is the entry point of the program. Given the goal `get-the-money`, the program will first assign tasks to `jim` and `dick` (bind `One` to one of them and `Another` to the other one), then will run the two agents concurrently. `One` will try to break into, while `Another` will try to provide him with the proper tool (actually I am too much animistic here; the actual mechanics is that the process running in the world `One` will be trying to prove the goal `break-into`, while the process running in the world `Another` will be trying to prove the goal `provide-tool-for`). According to the behavior specified in the world `robber` (or we may say "according to his nature"), `One` will climb the wall (which will take him some time), pick a `Safe` to break into, tell the safe type to `Another` through the shared variable `Safe` and wait for a `Tool`. After he gets the `Tool`, he will open the safe (again this will take time) and terminate. Meanwhile `Another` will wait until the safe type is known, pick the proper tool, send it to `One` and terminate. Finally the `time-condition` (unspecified here) will be checked and if they had a good timing, they will have `got-the-money`. Otherwise, OBJECT-PROLOG will backtrack and try other solutions (assign roles to robbers in another way, pick some other safe to break etc.).

So with a couple of primitive notions we wrote a 20-line program and then the description of its behavior took half a page, and it sounded like if the program was a quite intelligent entity (actually two entities). (Now, this may seem a vicious and illegal program, but maybe the police may use it to simulate different scenarios for breaking the bank and thus prevent them?)

However, too much freedom and flexibility may leave the user without the needed support for disciplined programming. The author clearly recognizes this (the picturesque words he uses to describe the problem are "As history has also proved sometimes—freedom without limits does not always lead to Paradise"). Another problem is efficiency, because these general mechanisms are quite expensive to execute. Both problems can be specialized by taming the model, specializing and limiting it in certain ways to fit particular user needs.

Although unsuitable in its present shape for industrial use, OBJECT-PROLOG is a most clear example of the power of relatively simple LP mechanisms to model intelligent behavior.

2.4 POKRS: Prolog-based Object-oriented KR System

POKRS[1] was developed in China in 1988. I have only a very brief description of the system, so I am including it here only for completeness. There is nothing special about this system (it is quite similar to ESP), except that when loading a logic program into a local knowledge base, automatic checks of the integrity of the inheritance hierarchy and checks for redundancy, subsumption and contradiction among clauses are performed.

The language used in the paper substantially hinders its understanding: a typical example is (sic) "The knowledge base organization is of taxonomically hierarchical architecture".

2.5 KSL/LOGIC: Reflexive and Extensible

KSL/LOGIC [10, 9, 8] is a LP extension of the OOP language KSL (which stands for Knowledge Specification Language), developed in the OWL programming environment of Electronic Data Systems Corp (EDS/OWL). An explicit goal in the design of KSL has been the possibility to integrate different paradigms into the language, so KSL/LOGIC may be seen as merely an example of one such integration. A fundamental means to achieve this paradigm-level extensibility is the *reflexivity* of KSL, which means that language constructs are represented as data objects in the language. So on the one hand, the language can operate on itself, which is most useful for the implementation of such tools as compilers, debuggers and performance profilers; on the other hand the language can be extended simply by addind new classes to the class hierarchy which describe the new language constructs. This reflexivity has been achieved much like in LISP: by severely restricting and making uniform the syntax of the language. Everything in KSL (method headers, variable declarations, statements etc.) is represented by some form of lists. Part of the KSL/LOGIC class hierarchy which shows that its language constructs are represented by classes is reproduced below:

```
LanguageObject
  BehaviorObject
    ClassBehavior
      ExtMethod
      SlotAccess
      Method
        PredicateBehavior
      ConstantBehavior
      VirtualSlotDefault
      VariableBehavior
      BuiltinPredicateBehavior
    TrapObject
  ExpressionObject
    MessageExpr
    VarAssignment
    VarReference
    ForExpr
    WhileExpr
    CommentExpr
    ConditionalObject
      LogicalExpr
      RelationalExpr
    ExistsExpr
    ForAllExpr
    PredicateExpr
    HornClause
    SwitchDomainExpr
    CutExpr
```

The following example which computes the age of a person should give a sense for the flavor of the language:

```
(#Method
  (&Selector Age)
  (&Class Person)
  (&ParmList (#List Self))
  (&VarList (#List BirthYear BirthMonth))
  (&ExprList (#List
    (Set BirthYear  (Year  (BirthDate $Self)))
    (Set BirthMonth (Month (BirthDate $Self)))
    (When
      (#List (GreaterThan $BirthMonth $CurrentMonth)
             (Subtract (Subtract $CurrentYear $BirthYear) 1))
      (#List True
             (Subtract $CurrentYear $BirthYear)
)))
```

Looks pretty much like LISP to me. The statements comprising the body of the method are called *expressions* in KSL, so this is the what `ExprList` comes from. They have to be explicitly packaged into a list, which is not quite connvenient. Method sends looks as follows: `Selector (Receiver)` and the lack of priority rules demands an excessive amount of parentheses. For me it is particularly disturbing that the same names have to be prefixed by various different "highly mnemonic" symbols in different contexts. The same method in SMALLTALK is substantially shorter and looks much better:

```
Person methodsFor: 'age'
  age
    "computes the person's age"
    | birthYear birthMonth |
    birthYear  := self birthDate year.
    birthMonth := self birthDate month.
    birthMonth > currentMonth
      ifTrue:  [^currentYear-birthYear-1]
      ifFalse: [^currentYear-birthYear]
```

An example of a predicate (computing the ancestor relation) in KSL/Logic is:

```
(HC (P ancestor %X %Y)
    (P parent %X %Z) (P ancestor %Z %Y))
```

(here `P` stands for "Predicate" and `HC` stands for "Horn Clause"). Again, the corresponding Prolog code is more appealing:

```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Definitely KSL needs a more user-friendly (actually programmer-friendly) compiler front-end.

The reader should note that the uniformity of KSL is of a quite different nature then the uniformity of Object-Prolog: the former is at the syntax and language construct representation level, while the latter is at the computational model level (because the computational model of KSL/Logic is completely different than the one of KSL).

The needed inference process (unification) is implemented by objects (the initial activation of a goal returns a search node object which is capable of performing backtracking and providing the alternative clauses which may unify with the goal). Although this is probably the easiest way to implement resolution in an OOP language and although the presence of such an explicitly represented proof tree (as opposed to one dynamically unfolded in time) may have some benefits for e.g. smarter search strategies, it suffers from inefficiency problems.

The integration of LP and OOP seems good both at the execution level (predicate behaviors can me freely mixed with procedural and/or functional behaviors) and the data level (predicate arguments can be not simply logic variables or syntactically constructed terms, but arbitrarily complex objects.

Despite its poor syntax, KSL/Logic solves a problem present in Orient84/K: the confinement of knowledge to the limits of one object. In KSL/Logic there is the notion of a *domain object*: an object which embodies *the whole* application domain. Of course, this object would be very complex if it was to do the whole work alone, but it is assisted by its subobjects which represent application subdomains. The predicate behaviors of the global domain object embody the most general and global relationships in the application domain. This idea seems innovative, however I fail to see why exactly the same could not be implemented in Orient84/K.

Probably it is curious to note that the two papers [10] and [9] overlap in at least 3/4 of their content, and the last 1/4 of [9] is included in [8], but the authors do not seem quite concerned about this: in [9] they refer to [10].

2.6 Agents: Cooperating KBS

Agents[7] is an architecture for KBS developed by Huang and Brandon at the University of Wales, Cardiff. (It is completely different from the Actors paradigm of Hewitt and Agha.) The initial motivation for its creation was support for intelligent CAD (manufacturing design) systems (though it can be used for other tasks as well, for example for modeling any decision-making process which in the real world is cooperative, e.g. doctors on a joint consilium). In engineering design there are numerous considerations to be taken into account and correspondingly, numerous different knowledge sources to be exploited. On the other hand, present day expert systems have to be focused in relatively narrow domain areas in order to be successful. This raises the issue of having a number of systems (agents), each of which is specialized in a certain domain area or aspect and all of them *cooperate* to solve the overall problem.

As an architecture, Agents is very similar to Orient84/K because it also adopts the notion of agents having local knowledge bases (and these may be even written in different private "proof languages"), which are packaged like objects and communicate with each other using a common "argument language" (I suppose that this is to be read "imperative"). It differs from Orient84/K in the way it is implemented: Orient84/K is based on a Smalltalk-like object-oriented model, while Agents is implemented by extending Prolog with OO constructs. The authors justify their approach by the claim that "Expertise has already been coded and accumulated in Prolog in many fields of engineering". My opinion is that the other approach is more appropriate; the authors point out themselves that there are two ways to see an Agents program: either as a set of agents or as a set of Prolog clauses which both provide predicate methods and the structuring of agents (and therefore the order in which these clauses are specified is important even for clauses with different predicate names):

```
agent(person).
  sub(student).  % subclass
  attribute(name).
  attribute(age).
  grandfather(X) :-  % a rule
    X ?- father(Y),  % send query to X
    Y ?- father(self).
endagent.
agent(student).
  super(person). % superclass
  attribute(courses).
endagent.
```

Such overloading of clauses with both structuring and operational tasks seems both unnecessary and inconvenient. (A feature of Agents can be seen in this example: sending a query to the agent `X` about its `father` in the rule for `grandfather`. Such sending a logical query to another object is typical of OO LP, but the most often used syntax is `X:father(Y)`.)

Another difference from Orient84/K is that in Agents object classes (which are called "agents", whereas object instances are called "objects" proper) are not implemented in the same way instances are (unlike Smalltalk). (And in addition there is some entity named "the meta-agent" which provides standard builtin predicates for the other agents). While this compromises the uniformity of implementation of the system, it may be beneficial for some specific KR purposes, as described below: objects carry only attributes and values, whereas agents in addition to attributes carry also methods and constraints. (Objects inherit features of agents through an instance inheritance, so they need not have all these features themselves.) Now the authors conjecture that a trace of such "lightweight" objects, with only partially specified or rawly estimated values, may serve as a history of the solution development (versions of agents) and facilitate explanation, solution revision and dead-end avoidance.

Another feature of AGENTS, named by the authors "demons, also called constraints", is that predicate methods may have "before", "after" and "if-failed" subparts, much in the spirit of method combination in FLAVORS and CLOS [14, Chapters 10.2.1 and 10.3] (the authors mention that demons are a more general notion including active values, and say that this method combination is only an example of what can be done in AGENTS). These auxiliary submethods have the following syntax:

```
head :- body.  % primary
head =: body.  % before
head := body.  % after
head -: body.  % if-failed
```

and the following semantics: the current goal is matched against the head of the primary method; if this succeeds than the "before" submethod is tried and if it unifies, it is executed; then the primary method is executed; then the "after" submethod is tried and eventually executed. If during this process the primary or "after" methods fail, then the "if-failed" method is executed. Therefore a goal fails if there is no clause for it in the current agent, or if the "before" submethod fails, or if the primary or "after" methods fail and the "if-failed" method also fails.[1] Some considerations regarding the inheritance of auxiliary methods and whether variable bindings imposed by their evaluation should affect the execution of the primary method, are discussed by the authors. Some of the goals may be marked as "askable" from the user: if the user answers "yes", s/he is further asked to supply values (if known) to the yet unbound variables in the goal; if the user answers "no" then normal PROLOG backtracking takes place (the user may also ask "why", see explanation facilities later on).

The communication between agents is accomplished using a blackboard. However the authors do not justify this design decision neither do they discuss how does this agree with the "message passing as only way of communication" paradigm of OOP. The blackboard is used to hold the "global state" of the problem solving process and the beliefs common to all agents. It may serve (but seemingly this has not been implemented) as a truth-maintenance medium (upon the detection of a conflict the participating agents are notified and advised as to revise their beliefs), as a task agenda for tasks yet to be performed, as an "advertising media" to display tasks for which there is no known agent able to complete them. Another use of the blackboard is to store a structured network of partial solutions (decisions) and their justifications which may be used in providing explanations and for intelligent backtracking. Clearly this design decision complicates the initial model: there is this new entity (in addition to agents) which serves as a quite active communication medium. Although not "wrong", this needs some justification. Isn't it possible to have an appointed agent serving as a "truth-maintainer" and "facilitator" for the communication of other agents (or even better, a number of such agents serving a number of small "interest groups")?

[1]Of course, these auxiliary methods are optional and may be omitted.

Now there is something unclear in the paper: on the one hand, the description of the language (which does not go to much more elaboration than the example given above) renders it quite simple and not very powerful. On the other hand, authors further describe every agent as a quite complete KBS with user interface, explanation facilities, knowledge acquisition facilities and so on.[2] While such model fits the title of the paper, clearly the language example given earlier is of a much lower level of granularity. The transition from this relatively low-level language to the complete KBS architecture is not described in the paper. How did the language facilitate the implementation of the architecture?

The explanation facilities supposedly included in every agent are not quite sophisticated: "why" explanations are given by dumping a restatement in English of the PROLOG stack from the current goal up to the topmost goal; "how" explanations are given by showing a more complete part of the proof tree, with leaf nodes marked as "fact" or "user input" or "procedurally computed". Although of unclear utility, these explanation facilities suggest that it may be necessary to keep parts of the proof and backtracking histories for this purpose.

The authors claim that the advantage of this system over ORIENT84/K is that "ORIENT84/K has complicated structure and therefore include many features that are difficult to master" [7, page 135]. Probably the authors here mean that it would be easier for a PROLOG programmer to adapt to AGENTS than to ORIENT84/K, but the reverse of this is that it would be easier for a SMALLTALK programmer to adapt to ORIENT84/K than to AGENTS, and furthermore the proper use of OOP requires a revision of one's programming habits anyway. I see the main benefit of AGENTS in its explicitly stated goal to support cooperating intelligent agents, and all other features described above as not so important. Such features are probably even inappropriate for a basic OOP+LP model for KR because they may be either not needed for a particular application or not sophisticated enough and thus useless for another one.

A very important idea put forward by the authors is that some KBS applications (for example engineering design) need multiview, multiversion, multicontext and multicomponent modeling facilities. Indeed, the history why a certain design decision has been adopted may be no less important than the decision itself, because the decision may not be directly reusable in a different design, while the underlying reasons can be reused with a higher probability.

3 CLASSIC/KL-ONE: A Different View

The work on KL-ONE started with the thesis of Ronald Brachman at Harvard in 1978 and still continues. As Brachman points out, since then there have been about 20 systems developed on the base of KL-ONE (mainly in Europe). Others who participated extensively in the development of these systems are Levesque and Patel-Schneider. CLASSIC and KL-ONE [3] expose a very different point of view than the languages described up to now. Their point of view is much more "KR oriented" than it is language oriented. As Patel-Schneider argues in [17], the Object-Oriented Programming Systems are exactly this—programming systems—and they are not very good in knowledge representation. He describes another class of systems—Object-Based Knowledge Representation Systems (OBKR)—which try to overcome some of the shortcomings of OOP systems for KR. The problem with OOP is that they are operationally based (for example described in the terms of their operation), and not representationally based. The main practical drawback from

[2]Does every agent in a system need separate user interface? I doubt so.

this is that objects in OOP do not have "knowledge" of their own purpose, thus the inheritance relations between them are to be specified manually. In an OOP system, where there is a hundred or a thousand of classes, this may not be a problem, but in a complex KR system which accounts for more subtle individual properties of objects, every object may need to have its own class and this may make the manual classification of these objects impossible. This objects' lack of awareness of their purpose in OOP limits its expressive power for KR because e.g. one cannot define a class axiomatically (by the set of properties it has), but have to create and attach it in the proper spot in the class hierarchy by hand. Even further, most OOP systems (excluding e.g. the languages Beta and Eiffel) allow properties of classes to be overriden down the class hierarchy (mostly because this is sometimes needed by "implementation inheritance" which purpose is code reuse), compromising its purpoose as a classification hierarchy.

It has been long ago argued by Hayes, Moore and Frisch that frames (or for that purpose, objects), have little value for KR (except for their structuriing properties), because they have no epistemological commitment to their purpose. For example, no distinction has been made between the definitional slots of frame (which make it to be of a particular class) and other "casual" slots.

So in CLASSIC and KL-ONE one of the most important modes of inference is *subsumption*: the automatic calculation of the inclusion relation between two sets of objects (two classes). In a sense, in OOP systems classes are represented by their *extensionals* (the set of objects comprising the class or at least a manually defined structure and behavior suite for the class), while in OBKR systems classes are defined by their *intensionals*: the set of properties that comprise the definiton of the class. Unfortunately, it is very hard to implement OBKR systems correectly and fully (some inconsistencies in KL-ONE-based systems were not exposed until recently) and some of their operations are intractable (computationally infeasible), at least in particular settings. As Patel-Schneider puts it [17] "Developing a usable OBKR system thus consists mostly of inventing a representational logic with the appropriate expressive and deductive power". An example of such a development is the use of a four-valued logic (with values true, false, unknown and contradictory), which localise some phenomena (e.g. a contradiction) so that they don't have to be propagated through the whole network of propositions.

CLASSIC and KL-ONE have *concepts* instead of classes, with *roles* instead of attributes, a *role taxonomy* instead of inheritance hierarchy, and *structural descriptions* which give an explication of the role of a concept in a particular relationship based on its connections to other concepts.

CLASSIC is based on LISP ("host" below refers to this implementation language) and as a language is very simple, its almost complete syntax is given below:

```
<concept> ::= THING | CLASSIC-THINF | HOST-THING | <concept name> |
              (AND <concept>+) |
              (ALL <role> <concept>) |
              (AT-LEAST <number> <role>) |
              (AT-MOST  <number> <role>) |
              (FILLS <role> <individual name>) |
              (SAME-AS <attribute> <attribute>) |
              (TEST-C <function> <argument>*) |
              (TEST-H <function> <argument>*) |
              (ONE-OF <individual name>+) |
              (MIN <number>) | (MAX <number>)
<individual name> := <symbol> | <string> | <number> |
'<host-language expression>
```

The advantage of such seemingly restricted language is that it has a clean compositional semantics which is required for the subsumption algorithm to be feasible.

Although very different from the systems described in the previous section, Classic and KL-ONE share with them two important features: the object-orientation for knowledge domain steructurization and the employment of logic (which in the previous systems is more LP) for computation.

4 Advanced Special-Purpose Logics

There is an abundance of non-classic logics for different aspects of the material world, developed by researchers in mathematics and philosophy, which wait for their application to KBS. These include Temporal Logic to represent time, different common-sense Physics Logics to reason about the physical properties of the world (some of these are not yet well formalised), Fuzzy Logic to deal with uncertainty, Epistemic Logic to reason about beliefs of intelligent agents, Autoepistemic Logic to reason about agents' beliefs about their own beliefs, Deontic Logic to capture the notions of permission, prohibition and obligation, Modal Logic about different possible developments of the state of affairs, Dynamic Logic to capture state change, etc. etc.

Some of these logics are plagued by paradoxes and inconsistencies, for some there are no computationally feasible models yet, some are not fully formalised, some will be very difficult to integrate but nevertheless I believe that they provide a very valuable layer of abstractions which should be used in KR tasks. Logic Programming is not only Prolog, it encompasses all attempts to implant formal logic theories into computing.

Although beyond the scope of this paper, I deemed these ideas significant enough to deserve a separate section.

5 Summary and Discussion

The paper surveys a number of OOP+LP languages for KR and contrasts them to the Classic approach. However my goal was not to determine which of the two is "the right" way to support a KR system, just to argue that LP augmented with the structuring features of OOP (or, conversely, OOP made more intelligent by the inference abilities of LP), is an appropriate way.

Another very important feature which is needed for knowledge bases is object persistence (one cannot have a knowledge base without first having a database). A lot of research in Deductive Databases has been performed by the LP community; Object-Oriented Databases have gone even further in the industrial aspect, but there seems to be a lag in their theoretical development (developments regarding the foundations of object updates and changing state have been undergone only recently). The first two conferences in the integrated area of Deductive *and* Object-Oriented Databases were held in 1989 and 1991. However this topic is out of the scope of the current paper.

So, as I see it, the ideal (at present) symbol-level *basis* for Knowledge Representation would be a Deductive Object-Oriented Database with versatile programming language, stable and well-founded formal semantics and non-classic logics (temporal, spatial, deontic etc.) for special-purpose modeling, extensibility for inclusion of other paradigms (without compromising the semantics), and possibilities for distributed computing. In my opinion, we are about five years from the time when something like this will be commercially available, and probably 15 years more until it is widely adopted in industry.

References

[1] Fu an Chen Yi-fen and Zhu. POKRS: A Prolog-based object-oriented knowledge representation system. In *1988 IEEE International Conference on Systems, Man, and Cybernetics*, pages 285–288, Beijing/Shenyang, China, August 1988.

[2] Daniel Bobrow. If PROLOG is the answer, what is the question? In *International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 138–145. ICOT, 1984.

[3] Ronald Brachman, Alexander Borgida, Deborah McGuinness, Peter Patel-Schneider, and Lori Resnick. The Classic knowledge representation system or, KL-ONE: The next generation. In *International Conference on Fifth Generation Computer Systems*, pages 1036–1043, ICOT, Japan, 1992.

[4] T. Chikayama. ESP–Extended Self-contained Prolog–as a preliminary kernel language of Fifth Generation computers. *New Generation Computing*, 1:11–24, 1983.

[5] T. Chikayama. Unique features of ESP. In *International Conference on Fifth Generation Computer Systems*, pages 292–298, Tokyo, November 1984.

[6] A. Doman. Object-Prolog: Dynamic object-oriented representation of knowledge. In T. Henson, editor, *SCS Multiconference on Artificial Intelligence and Simulation: The Diversity of Applications*, pages 83–88, San Diego, CA, February 1988.

[7] G. Q. Huang and J. A. Brandon. Agents: Object-oriented Prolog system for cooperating knowledge-based systems. *Knowledge-Based Systems*, 5(2):125–136, June 1992.

[8] M. S. Ibrahim and S. W. Woyak. An object-oriented environment for multiple AI paradigms. In *Second International Conference on Tools for AI*, pages 77–83, Herndon, VA, November 1990.

[9] Mamdouh H. Ibrahim and Fred A. Cummins. KSL/Logic: Integration of logic with objects. In *1990 IEEE International Conference on Computer Languages*, pages 228–235, New Orleans, LA, March 1990. IEEE Computer Society Press.

[10] Mamdouh H. Ibrahim and Fred A. Cummins. Objects with logic. In *Cooperation. ACM 18th Annual Computer Science Conference*, pages 128–133, Washington, DC, February 1990.

[11] Y. Ishikawa and M. Tokoro. Concurrent object-oriented knowledge representation language Orient84/K: Its features and implementation. In *OOPSLA'86*, Portland, OR, September 1986.

[12] Y. Ishikawa and M. Tokoro. Orient84/K: A language with multiple paradigms in the object framework. In *Nineteenth Annual Hawaii International Conference on System Sciences*, volume II: Software Track, Honolulu, HI, January 1986.

[13] R. Iwanaga and O. Nakazawa. Development of the object-oriented logic programming language CESP. *Oki Technical Review*, 58(142):39–44, November 1991.

[14] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley International Computer Science Series, second edition, 1990.

[15] Francis G. McCabe. *Logic & Objects.* International Series in Computer Science. Prentice-Hall, 1992.

[16] Hideo Miyoshi and Koichi Furukawa. Object-oriented parser in the logic programming language ESP. In *Natural Language Understanding and Logic Programming, First International Workshop*, pages 107–119, Rennes, France, September 1984. North-Holland.

[17] P. F. Patel-Schneider. An approach to practical object-based knowledge representation systems. In *Twenty First Annual Hawaii International Conference on System Sciences*, volume II: Software Track, Honolulu, HI, January 1988.

[18] M. Tokoro and Y. Ishikawa. An object-oriented approach to knowledge systems. In *International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 623–631, ICOT, Japan, 1984.