Mutable Object State for Object-Oriented Logic Programming: A Survey

Technical Report TR 93-15*

Vladimir Alexiev†
Department of Computing Science, University of Alberta
615 GSB, Edmonton, Alberta T6G 2H1
email: vladimir@cs.ualberta.ca

16 August 1993

Abstract

One of the most difficult problems on the way to an integration of Object-Oriented and Logic Programming is the modeling of changeable object state (*i.e.* object dynamics) in a particular logic in order not to forfeit the declarative nature of LP. Classical logic is largely unsuitable for such a task, because it adopts a general (both temporally and spatially), Platonic notion of validity, whereas object state changes over time and is local to an object. This paper presents the problem and surveys the state-of-the-art approaches to its solution, as well as some emerging, promising new approaches. The paper tries to relate the different approaches, to evaluate their merits and deficiencies and to identify promising directions for development.

Keywords: Object-Oriented Logic Programming, mutable object state, survey.

1 The Problem: Dynamics of Objects

From the research literature on integration of Object-Oriented Programming (OOP) and Logic Programming (LP) one gets the impression that the most obstinate difficulty on the way to such an integration is the following problem: how to represent (model adequately) state change in a declarative and logically sound way. On the other hand, the structural aspects of OOP (encapsulation, classes, inheritance, polymorphism and aggregation) can be accommodated in the LP paradigm with relative ease. Representation of state (object dynamics) and representation of structurisation (object statics) are largely orthogonal, but in some cases they interfere with each other. So notwithstanding that the emphasis of this paper is on mutable state (for a general survey of OOLP see the one by Davison [22]), in some cases I discuss static aspects as well.

Classical logic emerged from studies in the foundations of mathematics, and thus it is designed to reason about static things: relations, functions *etc.* Correspondingly predicate calculus adopts a notion of truth which is:

*Available by anonymous FTP from `ftp.cs.ualberta.ca: pub/TechReports/TR93-15/state.ps.Z` or `pub/oolog/state.ps.Z`. Comments and corrections are most welcome.

†Supported by a University of Alberta PhD Scholarship.

- Global: all the consequences of a logic theory are (conceptually) known to hold (this is named *logical omniscience*).
- Eternal (universal in time): a formula is either true or false, independent on the time it is evaluated. Furthermore, the same holds for terms (data items): a logic variable (as *e.g.* in PROLOG), once bound, cannot take on another value (except on backtracking). The property that a formula or a part of it always has the same meaning, no matter where and when it is evaluated, is called *referential transparency*.

To the opposite, object encapsulation and the dynamics of objects make object state:

- Local: the state of an object is its private property and no other object have access to it except by asking the owner using message passing.
- Time-dependent: in a system which should be able to model the dynamic world, object state changes over time.

This renders classical logic unsuitable for state change representation. There are a number of proposals for a logic of action and change, but none of them has ever become the core of a database or a logic programming system. On the other hand, classical predicate calculus is the foundation of *queries* in Relational and Deductive Databases and of Logic Programming, both in theory and in practice.

In addition to this general unsuitability of classical logic for the modeling of dynamic systems, there is a number of other "side" problems with the representation of change in LP which are better understood in the context of the respective approaches they appear in and are covered in the respective sections below. The following problem however is sufficiently general to be mentioned here: object-orientation is often used to implement open, highly interactive systems which do some input (perception), then some computation, then some output (actuation) and so on in a loop. Therefore those are *reactive* systems. Even if not open to the external world, a large object-oriented system consists of a large number of component objects and their *interactions*. Examples of such systems are user interfaces, operating systems, simulations, communication software, industrial control and robotics applications, *etc.* To the contrary, LP-based programs are *transformational*, in the sense that they get some inputs and produce (a set of) corresponding outputs which make a certain I/O relation true (one of the strengths of LP is that one can flexibly apply inputs; the program is like a black box with bidirectional pins which can serve as both inputs and outputs according to demand).

A complex object has a correspondingly large state information; an object in an OOLP program should store a number of its previous states for backtracking purposes. It is a very subtle matter to determine the exact amount of backtracking information (state history) to be stored between message sends. For example Conery in [19, p.433] writes: "Is there an operation analogous to cut that commits the system to a new object state and allows us to discard the history of some objects? How will this interact with the top-level user dialog?" The question is even more complex if there are more stimulus sources than a single user.

A sample proposal for the solution of this problem, which I deem *impractical* is as follows [73]: upon a message send a logical deduction process is initiated during the course of which every object computes its new state, but does not commit updates and responds to queries with its old state. After that a special notification is broadcasted globally and all objects move to their new computed

states. In other words, during the deduction process the database is "frozen" and moves to a new state synchronously, all at once. This convention greatly simplifies the logical semantics, but contradicts the very spirit of a distributed system of autonomous objects and introduces an unneeded distinction between "deduction" and "update"; also it is not clear what entity the programmer should "authorize" to broadcast the global "move to the new state" message and how this message will reach all its recipients.

The survey attempts to explicate the relations (sometimes even unintentional) which exist among the different proposals. The emphasis in this survey is on *efficient* implementation of state change, one which would be suitable for the lowest fundamental level of a general OOLP language. The following approaches are covered in the sections below: Assert/Retract, Declarative Database Updates and Transaction Logic, Modal and Dynamic Logics, Perpetual Objects, Logical Objects and Linear Objects, Linear Logic, Rewriting Logic and MaudeLog.

2 Assert/Retract: Non-Logical State Change

This section discusses a "cover under the carpet" approach to our problem, namely the approach of not addressing the problem at all. This approach is rather wide-spread in early integration proposals like the one by Zaniolo [79], and in proposals which treat objects as encapsulated labeled collections of clauses (local theories) like McCabe's $\mathcal{L}\&\mathcal{O}$ [59] (class templates) or Ishikawa and Tokoro's Orient84K [38, 39] (an object-based OOLP language). In the last two languages mentioned, an assignment operator is introduced for convenience and efficiency, but the logical semantics is the same as for `assert/retract`. Bancilhon in [9, p.19] talks about "clean queries and dirty updates" and argues that since all known systems which have a clear (fix-point) semantics do have a semantics only for queries but not for updates, we should accept this as a matter of fact and treat updates as destructive assignments.

Many of these approaches are based on the destructive modification of the logic program through the `assert` and `retract` operators of Prolog which insert and delete a clause from the program respectively. There are many problems with `assert/retract` which motivate *e.g.* Warren [72] to talk about "The evils of assert" and which make programming using these operators a rather non-declarative effort:

- They depend on the sequential execution of the literals in a goal (which is peculiar to Prolog), thus they do not respect the commutativity of logical conjunction.
- Theory change involves higher-order notions, which however with `assert/retract` remain hidden and implicit. Metalanguage (in which Prolog programs are written) and object-level language (in which Prolog talks about real-world objects) are mixed in a non-disciplined way. Since no distinction is made between the name of a formula (the argument of `assert/retract`) and the formula itself, this leads to strange effects such as:

  ```
  :- assert(p(X)).
  ```

  means "Assert that for all `X`, `p(X)` holds" (a *universally* quantified assertion), whereas

  ```
  :- p(X).
  ```

means "Does there exist an `X` such that `p(X)` holds" (an *existentially* quantified query).

- `assert` and `retract` do not undo their effects on backtracking, leaving an unwanted "residue" in the database. This makes it hard to use them for transaction programming where the changes should only be committed in the case when the transaction succeeds.

  There are PROLOGs featuring fully backtrackable `assert/retract`, *e.g.* OBJECT-PROLOG [24]. In SICSTUSPROLOG one can implement backtrackable `assert/retract` using a builtin meta-predicate `undo/1` whose term argument is executed on backtracking.[1]. For example backtrackable `retract` will be:

```
b_retract(X) :-
  retract(X),
  undo(assert(X)).
```

Another problem (which however is not inherent to `assert/retract`) is that changes are done globally to the database and are not encapsulated in a local part pertaining to a single object.

Although this approach is completely unsatisfactory, there are surprisingly many commercial OOLP systems which adopt it. This can be seen as a challenge to the academia to provide better techniques for modeling object state. For example the vendors of BIM-PROLOG say something to the effect that [31] "We could not use ineffective approaches based on maintaining complete histories of object states and we just could not wait for something better to develop".

3 Declarative Database Updates; Transaction Logic

One of the fundamental problems in the area of Deductive Databases is: how to incorporate new knowledge in the database, preserving the consistency of the updated database *(declarative database updates)*. The issue of dynamic DB updates becomes even more important with the development of Deductive and Object-Oriented Databases because of the inherent dynamic nature of objects. A lot of research has been performed in this area and one would expect that most of this research should be relevant to our problem. However, there are two potential misunderstandings which render most of the research inapplicable to our case:

- There is a difference between updating a declarative (deductive) database and updating a conventional or object-oriented database in a declarative way. The former is relevant in the area of Deductive Databases and it is the harder problem of the two, because it may involve updating general logic theories and therefore the whole suite of non-monotonic problems, *etc.* The latter is more relevant to conventional OOP and to mergers of OOP and LP, because usually object attributes (the dynamic parts of an object) are simple data terms and/or atomic formulæ. So the majority of the work in DB updates is *too general* to be applicable to OOLP: one would not like to deal with non-monotonicity issues for a simple change of an attribute which should be done very efficiently.

- There is a big difference between *updating* a database and *revising* it [51]. The former means changing the database in order to reflect a change in the real world, the latter means

[1] As pointed to me by Bernhard Pfahringer

incorporating newly acquired knowledge about a static world. The difference is that revision involves reassessment of the possibility of each of the models of the *old* theory, because some of them may turn out to be impossible in the light of the new knowledge; whereas update can never change the set of possible worlds of the old theory, it merely moves the system to a new set of possible worlds. For example, an inconsistent theory can never be made consistent using update [51], which is not the case if revision is used. Technically this involves a revision of the Alchourron-Gärdenfors-Makinson (AGM) rationality postulates for update operators. Informally, it means that the non-monotonicity issues involved are quite different for revision and for update, generally being simpler for update). Although many of the works are declared to be devoted to database updates, in fact most of them actually deal with revision.

With this precautions in mind, I turn now to a brief review of the declarative updates field.[2] Typical approaches which I deem impractical for the problem in hand include: allowing only additions but not deletions, with newer knowledge overruling inconsistent older knowledge [68]; maintaining a complete history of the database, as in the object-oriented version of Kowalski's Event Calculus [54, 53] developed by Kesim and Sergot [52], *etc.*

Two simple concepts predominate in the work in this field. One is the application of modal logic and the possible worlds semantics: changing a logical theory leads to a new (set of) model(s) for the changed theory. Another is the concept of *closeness*: the changed database should maximally resemble the old one, involving minimal change. Two variations are possible here: closeness can be interpreted in either a *syntactic* (minimal change of the set of formulæ), or a *semantic* sense (minimal change of the set of possible worlds). The work of Winslett [75, 76] is foundational in the sense that she first gave a generally accepted formalization of the notion of minimal change.

Fagin, Ullman, Vardi and Kuper [26, 25] has done some early work on updating databases with integrity constraints and the related problem of *view updates*: given a (possibly non-injective) mapping of the database state (a view), map an update of this view back to an update of the basic stored facts. An elementary (under a particular definition) update of a view may turn to be non-elementary for the stored database. Usually a number of views will be needed to disambiguate how to perform a basic update.

Abiteboul and Vianu [2] developed a number of declarative languages for database updates and obtained some results on expressiveness and complexity, but they were only concerned with the end result of executing an update, and not with the execution process itself. There are syntactic restrictions on the ways transactions can be constructed from simpler ones, and no transaction "subroutines" are allowed.

Chen [13] developed a calculus and an equivalent algebra for specifying and executing updates (restricted to insertion and deletion of single tuples), much in the tradition established for exploration of query languages.

A work which stands out with its very different approach to the problem is Transaction Logic ($\mathcal{TR}$) of Bonner and Kifer [10]. They opt not to deal with the whole spectrum of non-monotonicity issues related to database updates and assume that a *transition base* of elementary updates, specifying the set of possible transitions for each update, is given beforehand (usually through an algorithm which can enumerate it). In this way $\mathcal{TR}$ manages to be both very general (different models of updates can be accommodated in it), modular (a particular model of updates can easily be plugged

[2] Caveat: The paper is far from comprehensive in this respect. For more comprehensive surveys see [1] or [57]

in), and at the same time have a sound semantics (all the difficult non-monotonicity issues are encapsulated in the transition base and do not spoil the rest of the semantics). The formulæ of $\mathcal{TR}$ are database *transactions* built up from elementary updates which may succeed or fail, and $\mathcal{TR}$ focuses on their compositional properties. $\mathcal{TR}$ interprets all the classical logic connectives in terms of composition of transactions (*e.g.* conjunction constrains two transactions so that they both should run along the same execution path, disjunction corresponds to non-deterministic choice, *etc.*), and in addition introduces *serial conjunction*, denoted $\otimes$, which composes two transactions sequentially and its dual (under the de Morgan laws) *serial disjunction*, denoted $\oplus$. Also, left and right serial implications are defined

$$\begin{aligned} \psi \Leftarrow \phi &\stackrel{\text{def}}{=} \psi \oplus \neg\phi \\ \phi \Rightarrow \psi &\stackrel{\text{def}}{=} \neg\phi \oplus \psi \end{aligned}$$

which mean "Whenever ϕ happens, it must be immediately preceded (resp. followed) by ψ". In $\mathcal{TR}$ the difference between a transaction and a query is blurred, because a successful transaction may both do some changes to the database *and* bind some logic variables, thus returning an answer. This is corresponds well to the practice in LP, and is important for modeling methods in OOP where a method can both return a value and have side effects. Also, both hypothetical reasoning and committed transactions are possible in $\mathcal{TR}$. Other features which are important for database programming, like non-deterministic and bulk updates, non-deterministic sampling, static and dynamic integrity constraints on transaction execution, *etc.*, are also provided for.

A major advantage of [10] is that it develops a sound and complete proof theory of the *Horn* fragment of $\mathcal{TR}$ which is suitable for Logic Programming. It turned out that it is awkward to formulate the proof theory of *full* $\mathcal{TR}$ in $\mathcal{TR}$ itself, so a general logic of state change [11] is being designed for this purpose. The proof theory can both *reason* about transaction execution and actually *perform* this execution. Two dual proof systems are presented, for normal execution and for reverse execution (it is assumed that for every elementary update the transition base, given the current state, can compute both the successor state(s) and the predecessor state(s)). These two proof systems are profitably linked together for hypothetical reasoning ("What would happen if such and such update is performed").

A shortcoming of $\mathcal{TR}$ is that its model theory is somewhat complex: truth is defined on paths over states, each state being a *set* of possible worlds. Nevertheless, by focusing in the right problem (the properties of combining updates into transactions, not the properties of updates themselves), $\mathcal{TR}$ presents a rich and computationally meaningful framework for updates.

In summary, some of the approaches in this area seem too "heavy-weight" and concerned with *reasoning* about updates instead of per se *performing* updates to be useful for an efficient implementation at the lowest, fundamental level of a general OOLP language. Nevertheless these results are important to consider if a language closely tied with a Deductive and Object-Oriented Databases (resp. a database programming language), is seeked.

4 Modal and Dynamic Logics

There is a number of *logics of action* developed either in the Artificial Intelligence community or for providing a formal semantics of the execution of imperative programs. Those developed for AI applications (*e.g.* planning) are usually too general and cumbersome to be the basis of an

OOLP system, either because they represent actions as nested terms and reason explicitly about the current global state of the world (*e.g.* McCarthy's Situation Calculus and McCarty's Logic of Action), or because they focus on the temporal side of actions (*e.g.* Allen's logic of temporal intervals).

A number of formalisms for semantics of imperative programs stems from Modal Logic and Kripke's possible worlds approach. One of the first was Floyd-Hoare's program logic. Then comes Pratt's Process Logic [66, 34]. Here I describe Dynamic Logic [33] and its use for the semantics of object state change.

The language of Dynamic Logic consists of two kinds of expressions: formulæ $\phi, \psi, \ldots$ and programs $\alpha, \beta, \ldots$ (programs are deemed non-deterministic). Formulæ are constructed from simpler formulæ by the usual logic connectives, and in addition from a formula and a program by a couple of dual operators $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ which correspond to the usual modal operators "necessarily" $\Box\phi$ and "possibly" $\Diamond\phi$. (In fact Dynamic Logic can be seen as a kind of multi-modal logic, where modalities are formed by programs.) The semantics of these operators is defined as follows: if ϕ is a formula, α is a program and w is the current program state (world), then

$$\begin{array}{lll} w \models [\alpha]\phi & \text{iff} & w' \models \phi \text{ for every successor } w' \text{ of } w \\ w \models \langle\alpha\rangle\phi & \text{iff} & \text{there exist a successor } w' \text{ of } w \text{ such that } w' \models \phi \end{array}$$

Here "w' is successor of w" means that the program α leads (or may lead, if it is actually non-deterministic) from the state w to the state w'. Correspondingly, in $\phi \rightarrow [\alpha]\psi$, the formulæ ϕ and ψ are Dijkstra-style pre- and postconditions of α respectively.

Programs are constructed from elementary programs much like regular expressions using the following forms:

$\alpha;\beta$	sequential composition
$\alpha \mid \beta$	non-deterministic choice
α^*	infinite iteration (Kleene star)

The iteration construct α^* means "execute α zero or more times". The set of elementary programs usually consists of

$x := e$	assignment of a term to a variable
$\phi?$	test of a program-free formula

The test $\phi?$ succeeds when ϕ is true without changing the current state (the test is side-effect free), and aborts the program if ϕ is false (*i.e.*, there are no possible successor worlds).

Non-determinism turns out to be very useful, because *e.g.* an **if**-then-else construct can be defined as

$$\mathbf{if}(\phi, \alpha, \beta) \stackrel{\text{def}}{=} (\phi?;\alpha) \mid (\neg\phi?;\beta)$$

Sometimes Context-Free Dynamic Logic is considered (instead of Regular DL), where the language of DL is enriched with variables for programs (program names), programs are formed from elementary ones using a context-free grammar, and thus (mutual) recursion can be expressed.

Typical axioms for Dynamic Logic include

$$\begin{array}{rcl} [x := e]\phi & \equiv & \phi_{\{x/e\}} \\ [\phi?]\psi & \equiv & \phi \rightarrow \psi \\ [\alpha;\beta]\phi & \equiv & [\alpha][\beta]\phi \\ [\alpha \mid \beta]\phi & \equiv & [\alpha]\phi \wedge [\beta]\phi \end{array}$$

and the axioms of Modal Logic.

Although Dynamic Logic has been designed from the very beginning as a logic of imperative programs and state change, there is relatively small number of attempts to use it for the description of object state dynamics. In the work of Meyer and Wieringa [64, 74], Dynamic Logic is combined with Abstract Data Types and Order-Sorted Logic in a formal specification system called Conceptual Model Specification Language. ADTs (see also Section 8) are used to formalize structured object values, whereas DL captures object dynamics. The connection between the two relies heavily on the use of object identifiers. A similar series of works by Jungclaus, Saake, Sernadas and Hartmann [43, 42, 41] emphasizes dynamic aspects of object interaction and develops a language called OBLOG$^+$. Burandt [12] in his diploma thesis has started work in the same direction.

Similar work using multi-modal (but not dynamic) logic is one of Fariñas del Cerro and Herzig [28]. They use modalities of the form `ASSUME[p]` where `p` is a literal (propositional formula or its negation). This modality performs a transition to a world which is the same as the current one, except that the literal `p` is true in the new world. However their approach is too simplistic—only literal assertions/deletions are allowed, and the database is assumed complete, so that assuming ¬`p` amounts to deleting `p`—and their logic collapses to conventional propositional calculus. (This work may also be considered under Section 3 on database updates.)

Warren [72] and Manchanda [58, 57] in his thesis develop a theory of database updates in pure PROLOG, based on modal logic and the two operators `assume` and `forget`, which are similar to `assert` and `retract`, but have a clear logical semantics. The language OBJECT-PROLOG [24] developed in Hungary also features fully-backtrackable `assert/retract`. OBJVPROLOG of Malenfant, Lapalme and Vaucher [56] uses a similar approach.

Uustalu [71] proposes a two-dimensional modal logic which models uniformly two phenomena: overriding inheritance along the object hierarchy dimension and state change along the time dimension. Objects inherit the non-overridden part of their behavior from their ancestors in the inheritance hierarchy, much in the same way as they inherit the non-changed parts of their state from the previous instant in time.

A major problem with utilising Dynamic Logic for mutable object state is that it is designed to *reason* about programs and changes, not to actually *execute* them, or serve as the basis of an Object-Oriented Logic *Programming* system. In DL, programs and logical formulæ are two disparate types of entities, whereas in Logic Programming programs *are* formulæ, with computation equated to proof search. Also, in Object-Oriented Programming, queries (methods to return some information) and transactions (methods to perform some changes) can be freely mixed in methods which both return information *and* have side effects. So it is not by chance that the attempts to use DL for OOLP has come from the formal specification community and not from the logic programming community. Also, there are quite a few LP systems implementing modal logic [27, 29], and they are more theorem provers than programming systems.

5 Perpetual Objects: Exploiting the Dynamics of Proofs

As was mentioned in the introductory section, the notion of truth in classical logic is global and static. However there *is* a dynamic component in a LP or automated proof system: the proof process itself. A proof develops over time, and its (conceptually parallel) branching subproofs are spatially separated. Therefore, it is possible to represent objects as perpetually reappearing (by recursively calling themselves) predicates, bearing the object state in their argument terms.

There is a whole family of OOLP languages based on concurrent versions of PROLOG and the fore-mentioned paradigm. Concurrency (at least coroutine-based pseudo-concurrency) is needed in order to capture the simultaneous development of a number of objects. (But see Section 6 for two approaches achieving the same effect which do not rely on concurrency.) JosSome early ideas are proposed in Kahn's INTERMISSION [44], Hewitt and Agha's ACTORS also influenced this approach. The seminal paper is by Shapiro and Takeuchi [69] and later languages are MANDALA [30], VULCAN [47, 45, 48], POLKA [20, 21], $\mathcal{A}'\mathcal{U}\mathcal{M}$ [77, 78]. The base language of the Fifth Generation Computer Systems project ESP [14, 15, 40] can also be counted here. A good introduction to these ideas is [46].

The mechanics of modeling OO notions in this paradigm is as follows [69]:

1. Objects are represented by perpetual predicates (proof processes). In order to persist between message sends, the object has to explicitly reinstate itself by making a recursive call.

2. Object state is represented by the arguments of the object predicate. State change is achieved by substituting in the recursive call values differing from the input parameters of the message send.

3. One of the arguments of the predicate is a message stream to the object, represented as a lazy list (one whose tail can be undetermined). After handling a message, the object passes the rest of the stream to itself:

```
object (State, [mesg | StreamRest]) :-
  change (State, NewState), object (NewState, StreamRest).
```

4. The object process is activated when a message is received (the head of the stream is bound), and after handling the message the process is suspended. Technically this means that the recursive call is not pursued immediately, but is postponed until (the head of) `StreamRest` gets determined; and also that no backtracking information is stored for this call (that is, tail recursion elimination is being done). In committed-choice languages the clause body is being split into a guard and body proper: `Head :- Guard | Body`, and after the guard is satisfied, execution commits to this clause by forgetting all other choice points (`|` is called the commit operator). So our earlier example becomes

```
object (State, [mesg1(Param) | StreamRest) :-
  this_is_the_correct_method1 (mesg, Param) |
  change1 (State, NewState), object (NewState, StreamRest).
object (State, [mesg2(Param) | StreamRest) :-
  this_is_the_correct_method2 (mesg, Param) |
  change2 (State, NewState), object (NewState, StreamRest).
```

5. If an object cannot handle a message, it delegates it to its *superobject* (an acquaintance held in one of the predicate parameters), similar to the ACTORS paradigm and SELF class-less prototype objects.

6. A message is responded to either by sending a dedicated message in the opposite direction or by binding a place-holder "result" variable built-in the message term. This way the sender

does not have to wait until the message is responded to, but can continue execution until the result is needed, at which point it will be suspended if the result is not bound yet.

A problem with this approach is that the semantics of CONCURRENT PROLOG and other committed-choice languages is very different from the semantics of PROLOG (some authors even go as far as to call CONCURRENT PROLOG an "impure dialect" of PROLOG). The (potential) non-determinism of PROLOG's Selection Rule (which clause to try next in order to achieve the current goal) is a kind of "don't know" non-determinism and does not affect the declarative semantics of the program because backtracking tries all applicable clauses in turn (unless `cut` is used). However in committed-choice languages after a guard is satisfied the execution commits to the corresponding clause and the execution of all other clauses is abandoned. Therefore it is an essential fact that the guards are tested *before* the body of the clause is executed, and so the commutativity of logical conjunction is not respected. Furthermore, the programmer should ensure that it does not matter which of the eligible clauses will be committed to ("don't care" non-determinism).

The fore-mentioned problem, although making programming in CONCURRENT PROLOG a less declarative effort than one would like it to be, has a bright side: it makes the programming of reactive systems (as described in Section 1) possible, because the search tree is pruned early and not all possible answers to the stimulus are computed, but only the ones relevant to the current state (of course, this can be achieved in conventional PROLOG, but it will involve extensive use of `cut`).

Another problem is that all the arguments of the predicate (attributes of the object) are to be passed to the reinstating recursive call, even if only a small part of them are changed (similarly to the frame problem in AI). It is possible to overcome this by packaging all attributes in an Abstract Data Type entity which is only capable of performing elementary attribute updates, but this introduces another level of indirection and thus, inefficiency.

Another problem is that this approach does not model very well the structurisation (static) aspects of OOP. In order to achieve incremental (default) programming (specifying only the methods which specialize an object from its corresponding superobject), the programmer has to maintain in the subobject a pointer to its superobject and call the superobject explicitly. It would be fair to say that this approach achieves delegation only, but not inheritance. Therefore, plainly applied, this approach is rather low-level. Some higher-level languages (*e.g.* VULCAN and $\mathcal{AUM}$) are implemented as preprocessors which translate them to an underlying concurrent LP language, or as specialized interpreters, which perform delegation automatically. But even if these languages are more convenient, they are somewhat inefficient compared to convenient OOP languages like C++ or SMALLTALK, because the representation of an object is not a record formed by appending the specializing attributes of the object to the record of the superobject, but rather a chain of partial records connected by delegation links. So in order to use an inherited attribute, a method in the object has to ask its superobject explicitly. Also, since a `self` pointer pointing to the object is not passed automatically upon delegation, polymorphism is problematical. Namely, if the superobject needs to call a polymorphic method in the object, an explicit passing of `self` is to be arranged by the programmer. This also creates a proliferation of processes corresponding to partial objects and complicates the message patterns between them.

Another problem is that the `is-a` and `part-of` hierarchies are mixed-up: both require that the object holds pointers to its parts/superobjects and both rely on explicit delegation. This has both bad methodological implications and causes inefficiency. In SMALLTALK the parts of an object are represented by pointers in its record (unless they are atomic non-object entities, like numbers), but

in C++ one has the option to either have pointers to the subparts, or package the part objects themselves in the record.

This approach models relatively well the short-term aspects of state, but is not that good at modeling long-term aspects. For example, it is not quite clear how to integrate it with an Object-Oriented Database, short of using completely disparate object representations in the short-term and long-term memories.

The inconvenience that the programmer should write explicit code in order to conform to the object-oriented style has a reverse side: it is possible (and even not very hard) to program very flexible and varied patterns of communication and delegation, which would be hard if a commitment is made to specific patterns in the language itself. For example an object may have a number of message streams (ports) and not only one; one-to-many broadcasting is easy and many-to-one communication is possible by introducing special stream-merging components or a generalization of streams called channels. The reader will see a similar phenomenon—inefficiency and/or inconvenience, but on the other side great flexibility—in other approaches as well (Section 6).

Notwithstanding its deficiencies, this approach is an easy-to-implement integration of completely separate ideas which are fitted very well together.

6 Logical Objects and Linear Objects: Multiple Heads

It turns out that the salient feature exploited by the perpetual objects approach (covered in the previous section) is not concurrency itself, but the ability to pursue more than one goal simultaneously, thus modeling the simultaneous development of more than one object. Concurrency (or coroutine-based pseudo-concurrency) is only a means to this end; there are other ways to achieve it. One of them is to extend Prolog by allowing many heads in a clause and/or disjunctive goals. In order to avoid the combinatorial explosion of the search space, restrictions should be imposed on the way these additional heads are pursued.

This section describes two proposals which, although stemming from different grounds and having different formal justifications, nevertheless end up in a quite similar form.

Logical Objects was proposed by Conery [18, 17, 19] in 1987. It introduces a new kind of literals—*object literals*—whose arguments carry the object state (similarly to Section 5). However the thread of control is not programmed as a message stream held by the object, but more in the spirit of conventional Prolog using "normal" literals (Conery calls them *procedure literals*). If the current goal contains an object literal, it is not pursued independently, but is "consumed" only in conjunction with other procedure literals. (An exception to this rule is that when the goal consists of object literal(s) only, they are executed by themselves, which corresponds to some "finalizing" actions with the object(s): destroying them or checking their integrity.)

A program clause is allowed to have in its head, in addition to one procedure literal, zero or more (but usually 0 or 1) object literals. Since the head is deemed a conjunction of literals, these are not really clauses anymore (which are disjunctions of literals). However this does not change the inference method drastically (it is very similar to normal binary resolution), because an object literal is only pursued together with a procedure literal. Thus the proof that some object exists is constructed in parallel with the proof that it has certain properties.

In order for a clause to fire, all its heads are to be matched in the set of goals, then these goals are consumed and the clause is executed.

```
object(ID,State), mesg :-
  change(State,NewState), object(ID,NewState).
```

(contrast this with the example in Section 5). The merit of this approach is that is does not depend on a CONCURRENT PROLOG implementation: the language HOOPS described in [19] is normal backtracking PROLOG. Object state in the CONCURRENT PROLOG approach is held in a suspended (waiting for a message) process, while in this approach the object literals in the goal are suspended ("pushed back" in the set of goals) until a suitable procedure literal is available. Therefore suspending here is performed globally, in only one place.

If there are no object literals in the head of a clause, but there is some in the body, a new object is created. Deletion of objects is modeled by having an object literal in the head, but no object literal in the body. Object literals usually bear an OID used to distinguish among separate instances of the same class. If there are two object literals with the same ID in the head and in the body, this corresponds to state change.

```
new_stack(ID) :- generate_id(ID), stack(ID,[]).  % creation
push(ID,X), stack(ID,S) :- stack(ID,[X|S]).      % update
pop(ID,X), stack(ID,[X|S]) :- stack(ID,S).       % update
top(ID,X), stack(ID,[X|S]) :- stack(ID,[X|S]).   % pure query
delete(ID), stack(ID,_).                         % destruction
```

Conery does not develop any inheritance mechanisms in his original proposal, he only observes that "The LOGICAL OBJECTS approach does not *hinder* the implementation of inheritance". A later implementation in ANDORRA at SICS [16] proves him right. (ANDORRA is a concurrent LP language, but it does not use commitment: a choice point is delayed until sufficient information to make the choice is available.) If the object A inherits from the object B, then A consists of two object literals with the same ID, one for the base object B and one for the additional attributes of A. An example adapted from [16] follows: assume that we have an `account` object and we extend it to a `tax_account` object which also knows about tax deductions and accumulates the current deductions `D`.

```
(1) new_tax_account(ID) :- new_account(ID), tax_account(ID,0).
(2) tax_account(ID,D), spend(ID,X) :-
(3)   spend(ID,X),
(4)   deduction(X,D1), tax_account(ID,D+D1).
```

What appears to be a misplaced recursive call in `(3)` is in fact a call to the overridden method in the superclass. The following mechanism accomplishes this: when the head of the clause `(2)` is matched in the set of goals, the literal `tax_account(ID,D)` is consumed. Therefore the predicate call `(3)` cannot match again the same clause `(2)`, because no corresponding object literal is present. This calls the supermethod and upon return from it, the needed additional computation is performed and the `tax_account` is re-established. However this technique is order- and implementation-dependent, because the application of a clause should be non-atomic. Furthermore, Conery and Haridi does not explain how is the more specific method chosen for execution in the first place, if the literal `account` needed for the execution of the more general (super-) method has also been present. This inheritance mechanism (as well as this problem) is very similar to the one used in LINEAR OBJECTS (see below), which has been developed earlier.

There is an efficiency problem with this approach: the concept of message sending in LOGICAL OBJECTS is quite far from message passing in traditional object-oriented languages. A sender does not really send the message *to* the receiver, it rather includes receiver's ID in it and then "posts" the message to a global blackboard-like structure (the set of goals), from where the receiver picks it using pattern matching. Conery and Haridi in [16] mention "Pattern Matched Object Selection" (PMOS) and argue that it simplifies programming (compared to the CONCURRENT PROLOG approach), because no explicit communication patterns are to be established, no streams are to be connected *etc.* Although this observation is true, PMOS has bad influence on efficiency, because a general pattern matcher (*e.g.* of the kind of RETE) is to be employed. Object IDs in LOGICAL OBJECTS are not machine-oriented effective address-like entities (currently they are simply integers), and they *cannot* be: during the processing of every message the object is consumed and then recreated again, and it would be impossible to notify all objects who reference it about this "change of address". SMALLTALK has a similar problem (objects may be moved during garbage collection), which is solved using two redirection levels. However even if this is applied here, the problem with the distribution of the object over a number of partial records (corresponding to incrementally extending the object during its specialization) remains. The unstructured blackboard-like global object space reappears in MAUDE (see Section 8), and the fragmented object records and the lack of "real" OIDs is even worse in LINEAR OBJECTS (see below).

LINEAR OBJECTS of Andreoli and Pareschi [8, 6, 7] goes one step further than LOGICAL OBJECTS: not only is an object separated from its message stream, but the object literal itself is split into small pieces each bearing only one attribute or a few related attributes.

Note: the LINEAR OBJECTS system, which is based on Linear Logic, probably belongs to Section 7, but I wanted to emphasize its similarity to LOGICAL OBJECTS. In any case, it cannot be fully understood without reading the section on Linear Logic. More papers are available as technical reports from the European Computer industry Research Center (ECRC) in Germany.

This finer granularity allows methods to specify and carry-over only the "essential" attributes of the object: the ones which are either changed by the method or are inputs to the method. It also allows the system to "infer" *is-a* relations automatically: an object A *is-a* B iff the multiset of attributes of A is a superset of those of B. This justifies Andreoli and Pareschi in saying that LINEAR OBJECTS have "built-in inheritance". For example all the methods for `point` below (*e.g.* clause `(5)`) also apply to a subclass "colored `point`". In the methods of the specialized class (*e.g.* `(6)`), one does not have to mention and carry-over attributes from the base class (*e.g.* `x(X)` and `y(Y)`), unless they are really needed.

```
(5) point @ x(X) @ y(Y) @ move(X1,Y1) <- point @ x(X1) @ y(Y1).
(6) point @ color(C) @ set_color(C1) <- point @ color(C1).
```

The new connective `@` in the clauses above is the *multiplicative disjunction* of Linear Logic (see Section 7). It is used as the "glue" which ties the part of an object together. The other connective (which is allowed in clause bodies, but not in clause heads) is `&`: *additive conjunction*. It aggregates objects and messages into larger entities called *contexts*, and at the same time separates objects from one another and does not allow parts of different objects to mix together (`@` binds stronger than `&`). The connective `<-` which divides the head of a clause (method) from its body is *linear implication*.

Similarly to a probelm in Conery's proposal, it is not clear how exactly the most specific method

(overriding a method in a superclass) is chosen for execution. There is one more kind of implication: `<=` (linear implication combined with the modality *of course*) which specifies that the method is applicable only if the literals in the goal *exactly* match the literals in the clause head (and are not simply a superset), but Andreoli and Pareschi do not use it for this purpose.

LINEAR OBJECTS are well suited for concurrent programming, because additive disjunction introduces a kind of OR-parallelism, dual to the usual AND-parallelism of goals in a clause. Andreoli and Pareschi compare these to the internal distribution of tasks in an organization and the external co-operation among organizations. (In fact LINEAR OBJECTS uses message *streams*, but this was of no importance for the examples `(5)` and `(6)`).

The problem of LOGICAL OBJECTS that an object-message pair is to be pattern-matched to its corresponding method reappears here, but it is even worse, because the granularity is finer. This finer granularity enables great flexibility, but one has to pay for it. Andreoli and Pareschi have used partial evaluation techniques to diminish this problem; probably the experience gained from the SELF language will be useful here.

An "intuitive" criticism of LINEAR OBJECTS is that such "finely crushed" objects do not seem very well encapsulated, their only "capsule" is the surrounding pair of `&` connectives. This leads to bad consequences, *e.g.* all the attributes of an object have to be mentioned for a destruction or a copy operation, thus every object should have its own such operation. In a conventional OOP language the compiler takes care of this. Also, a method in LINEAR OBJECTS does not belong to a distinguished class explicitly by the program; its owner is only determined at runtime by the set of object literals which the method lists in its head.

In a summary, LOGICAL OBJECTS and LINEAR OBJECTS are refinements of the objects-as-processes approach (Section 5) overcoming many of its deficiencies, mainly in the structurisation and inheritance aspects. However deduction in them heavily uses pattern matching of object-message pairs to methods, which may be a source of inefficiency. Also, "real" object identifiers are impossible in these approaches. LINEAR OBJECTS are firmly founded in Linear Logic and thus have a sound and well-understood semantics.

7 Linear Logic: Resources Rather Than Truth Values

Linear Logic has been proposed by Jean-Yves Girard in 1987 [32] and since then has received the attention of many computer scientists (see *e.g.* [3] for some developments and [4] for a survey). The reason for this is that unlike classical logic, Linear Logic regards propositions as *resources* which are consumed and produced during the inference process, and not as universally valid (or universally false) assertions. This is why Linear Logic is called "resource-aware" and why it is useful in many areas of computer science which deal with resources. Classical logic treats the proof process only as a device to achieve some conclusions, whereas in Linear Logic the proof process is a "first class citizen" and is no less important then the conclusions themselves. This is why it is useful to describe and program (concurrent) *processes*. (Of course, classical logic has rich and substantial proof theory, but it is *outside* the logic.)

There are many ways to explain *why* Linear Logic, but one of the most natural ways is to look at it as restrictions on the allowed proofs in order to make them more constructive [5]. A proof in classical logic contains many redundancies which increase the space of proofs a lot and make searching for proofs hard. These include various non-essential choices (permutations of parts of the

proof) and, worse of all, dead-ends—subproofs of propositions which are then simply thrown away. Linear Logic gets rid of these using two approaches:

- Every proposition should be used once and exactly once during the proof (resource-awareness). The sequents of Linear Logic are not sets of formulæ, but multisets. Linear Logic does not have the usual "structural" rules of classical logic:

$$[\text{Weakening}]\ \frac{? \vdash \Delta}{?, \Phi \vdash \Delta} \qquad [\text{Contraction}]\ \frac{?, \Phi, \Phi \vdash \Delta}{?, \Phi \vdash \Delta}$$

  which allow one to reuse or throw away formulæ($?$ and Δ denote multisets of formulæ, Φ and Ψ denote individual formulæ). (But a formula may be preceded by the modal *exponential* operator *"of course"* ! or its dual *"why not"* ?, in which case it can be used any number of times.)

- Linear Logic uses special syntactic devices in order to explicate the intended proof of a formula. In other words, the syntax guides the proof [5, p.27 ff].

The idea of syntax-directed proof search leads to splitting conjunction and disjunction into two forms, *additive* and *multiplicative*:

	additive	multiplicative
conjunction	$\&$	$\otimes$
disjunction	$\oplus$	@, $\wp$

These new connectives are not idempotent (*e.g.* $\Phi\&\Phi \not\equiv \Phi$), unlike the classical connectives (*e.g.* $\Phi \wedge \Phi \equiv \Phi$), so a formula cannot be duplicated or disposed of arbitrarily. This makes it possible to differentiate between two intended uses of the connectives which are mixed in classical inference rule systems. Each of the linear connectives have one corresponding inference rule, for conjunctions:

$$[\&]\ \frac{\vdash ?, \Phi \quad \vdash ?, \Psi}{\vdash ?, \Phi\&\Psi} \qquad [\otimes]\ \frac{\vdash ?, \Phi \quad \vdash \Delta, \Psi}{\vdash ?, \Delta, \Phi \otimes \Psi}$$

(both of these rules would be allowed for $\wedge$ in classic logic). Analogously for disjunctions:

$$[\oplus]\ \frac{\vdash ?, \Phi}{\vdash ?, \Phi \oplus \Psi} \qquad [@]\ \frac{\vdash ?, \Phi, \Psi}{\vdash ?, \Phi @ \Psi}$$

Linear Logic is complicated compared to classical logic. Not only it has more connectives, but they are also non-functional, which means that truth tables cannot be used. For example the linear negation $\phi^{\perp}$ of a propositional letter ϕ is not reducible to ϕ, in a sense it is independent of it. Full propositional linear logic is undecidable (classical propositional calculus is decidable; it is a special case of propositional linear calculus with every formula preceded by !). The complexity results for fragments of Linear Logic are also not very encouraging; for example the multiplicative fragment $\otimes$, @ which corresponds to Horn clause programming, is NP-complete [49, 50].

During goal-directed proof search inference rules are used backwards (the conclusion is given and we are searching for the premises). The specific proof-theoretic properties of Linear Logic justify that one can apply initially only the four pure logic rules listed above until the goal is split

to atoms, and use the logic program only afterwards. This simplifies greatly the proof construction process.

Andreoli and Pareschi use in LINEAR OBJECTS the connectives @ and & and employ the specific property of inference rule [&] that the "context" ? is duplicated in the two premises for "object cloning".

Linear Logic has been applied in various areas of computing science, from encoding Petri Nets in logic to optimizing functional languages by controlling interference between expressions. There is a relatively small but growing number of attempts to use it for logic programming: [5, 35, 37, 65, 70]. I believe that it can be used more "locally" to deal with mutable object state, not in the particular style of LINEAR OBJECTS, in order to avoid the global blackboard space, to avoid splitting objects into such small pieces, and to achieve an OOLP language more faithful to traditional OOP.

8 Rewriting Logic: Free Object Reductions

In 1990 José Meseguer proposed a logical theory of concurrent objects [60, 61] based on his earlier joint work with Joseph Goguen on Abstract Data Types and the OBJ family of equational languages. He defines Rewriting Logic and a language for declarative concurrent object-oriented programming called MAUDE. Later [62, 63] he develops a theory of general logics based on Category Theory in order to formalize the notion of a Logic Programming Language and a methodology of integration of such languages by mapping them to a richer logic which encompasses them all. (Meseguer calls "Logic Programming Language" any declarative in nature programming language and what traditionally is called LP he names "Relational Programming".) He proposes Rewriting Logic as such an encompassing logic and using it integrates rigorously the functional, relational and object-oriented paradigms. In this work he extends MAUDE to a language called MAUDELOG which includes Horn clause programming. Some aspects of the paper [63] are rather technical, but overall it is very clear and enlightening.

Meseguer argues that declarativeness is only one of the advantages of LP, and another no less important is its suitability for *concurrent programming*: logical axioms bear no inherent order or sequence, so their application can be performed in parallel. However Horn logic (and more generally classical logic) are not suited well for dynamic computations, because they are based on a Platonic, static notion of truth and they deal with static objects: functions, relations, *etc.* This makes traditional LP languages awkward for concurrent programming and even for sequential object-oriented (state-oriented) programming and therefore unsuitable for large programming tasks.

Rewriting Logic is very similar to (order-sorted) equational logic, but in addition to equations one can specify *rewriting rules*, which differ from equations in that they work only in one direction (the corresponding relation is transitive but not symmetric). An example is

```
(1) eq d =  q+q+q+q .
(2) rl d => q q q q .
```

Here `d` stands for "a dollar", `q` stands for "a quarter", the equation `(1)` specifies that a dollar equals four quarters, and the rewrite rule `(2)` may specify the behavior of a change machine which can break a dollar into four quarters, but not the other way around. Lets add the appropriate declarations in order-sorted logic (assuming that the sort `Nat` is imported from some library module):

```
(3) sorts Cents Purse .
```

```
(4) subsort Cents < Nat .
(5) subsort Cents < Purse .
(6) ops d q : -> Cents .
(7) op _+_ : Cents Cents -> Cents
      [assoc comm id: 0].
(8) op __  : Purse Purse -> Purse
      [assoc comm id: null].
```

Here `(3)` declares the sorts `Cents` for an amount of money and `Purse` for a set of coins, `(4)` declares that `Cents` is a subsort (specialization) of `Nat`, similarly `(5)` says that a single coin makes a purse. Line `(6)` declares `d` and `q` as constants of sort `Cents` (zeroary functions returning `Cents`). Line `(7)` declares the operation Plus which adds two amounts of `Cents` and is associative (this justifies the absence of parentheses in `(1)`), commutative and with `0` as its identity element. The two underlines `_` around the `+` declare it as a binary infix operation. Line `(8)` declares an empty-syntax operation (there is nothing between the two `_`) which again is associative, commutative and with identity element `null` (`null` is what stays between the `d`'s on the right-hand side of `(2)`: just nothing). The operation in `(8)` is simply multiset union (denoted by juxtaposition of elements) which takes two `Purse`s and combines them into a bigger `Purse`. Now it becomes clear that the sort `Purse` denotes a multiset (bag; that is, duplicates are allowed) and not just a plain set, because the union operation is not idempotent.

Note: Semantically the two operations `(7)` and `(8)` are rather different: the former takes two elements and forms their sum, while the latter keeps the two elements distinct and only puts them into an aggregate. We may define a function which sums all the coins of a purse:

```
 (9) op amount : Purse -> Cents .
(10) vars P1 P2 : Purse .
(11) eq amount(null) = 0 .
(12) eq amount(P1 P2) = amount(P1) + amount(P2) .
```

but once two amounts of `Cents` are summed, it is impossible to recover the coin distribution of the result.

In MAUDE an object is represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object identifier, a term of sort `OID`; C is the object class; $a_1, \ldots, a_n$ are the object attributes and $v_1, \ldots, v_n$ are their values (the object state). Messages are represented similarly as terms bearing the identity of the receiver and other relevant information. These message parameters are "injected" in the message term by corresponding message constructor functions, *e.g.*:

```
(13) msgs credit debit : OID Nat -> Message .
(14) msg  transfer_from_to_ : Nat OID OID -> Message .
```

Line `(13)` gets a receiver's OID and an amount and constructs a `Message` for the corresponding operation. Line `(14)` declares a ternary mix-fix operation which constructs a message to transfer a `Nat` amount of money from one account to another. For type-checking purposes it probably would be better to declare the object parameters of these functions not simply as `OID`, but as specific

object classes, *e.g.* `Account.OID`. There is no problem to have separate OID types for every class and a preprocessor which does this automatically has been implemented for MAUDE.

The technical device of an Associative Commutative operation with Identity (ACI-operation) is used in Rewriting Logic to represent collections of distributed objects *and the messages flowing between them*. The ACI operation binds these entities into an object space (configuration of distributed objects):

```
(15) subsorts Object Message < Configuration .
(16) op __ : Configuration Configuration -> Configuration
       [assoc comm id: null] .
```

When an object needs to communicate with another object, it simply posts a message to this global space. Then this message interacts with the receiver as governed by the appropriated rewrite rules: the left part of a rule is matched against the message and (part of) the state variables of the receiver and then the message is (usually) consumed (rewritten to `null`) and the receiver's state changed. Also, another message can be generated during this process and sent to some other object.

Unlike *e.g.* SMALLTALK, where a method always belongs to a particular class and every message should have a designated receiver (so $2 + 3$ is interpreted as the message $+3$ sent to the object 2, which is rather unnatural), in MAUDE this need not be so. The message `transfer_from_to_` has *two* receivers in the sense that the corresponding method (rewrite rule) should be able to locate both objects `(19)` and `(20)`:

```
(17) rl
(18)    transfer A from X to Y
(19)    < X : Account | bal: XB >
(20)    < Y : Account | bal: YB >
(21) =>
(22)    < X : Account | bal: XB-A >
(23)    < Y : Account | bal: YB+A >
(24) if XB >= A .
```

(of course, this can be written more compactly). This rule says "If the entities `(18)`, `(19)` and `(20)` are simultaneously present, they can be rewritten to `(22)` and `(23)`, provided `(24)` holds" (so we have *Conditional* Rewriting Logic here). Please note that `(18)` is not a "method name", it is an entity which should be present in the `Configuration` for this rewrite rule to fire, just like the two receiver objects `(19)` and `(20)`.

Formally the difference between equations and rewrite rules is not that big: the former work in both directions (symmetric), whereas the latter work in only one direction. Actually in the operational sense they are almost the same, because one usually uses equations unidirectionally: to rewrite function calls with the corresponding function definitions in order to obtain a particular canonic form. However the underlying logical semantics are rather different, the unidirectional nature of rewriting rules corresponding to the unidirectional flow of time in an evolving distributed system.

Adding relational programming (going from MAUDE to MAUDELOG) is not hard: it is well known that PROLOG goal-directed deduction can be thought of as rewriting the current set of goals using the program clauses as rewriting rules until the empty goal is reached. Logic variables can be modelled by allowing variables in the rewriting rules, and non-determinism (with which

PROLOG deals by backtracking) is accounted for by the non-deterministic nature of rewrite (there may be many rules which match the current configuration). (A similar in spirit work by Debart [23] demonstrates clearly the power of equational rewriting techniques by implementing multi-modal logic programming through a translation of modal formulæ to many-sorted equational formulæ.) For example the PROLOG program

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
parent(peter,paul).
parent(mary,paul).
```

is translated to the MAUDE module

```
mod FAMILY is
  extending PROLOG .
  sort People .
  ops peter paul mary : -> People .
  ops parent grandparent : People People -> Bool .
  vars X Y Z : People .
  rl parent(X,Y), parent(Y,Z) => grandparent(X,Z).
  rl true => parent(peter,paul) .
  rl true => parent(mary,paul) .
endm
```

The module `PROLOG` imported by `FAMILY` defines the sort `Bool` and the ACI operation `_,_ : Bool Bool -> Bool` (conjunction) with identity `true`. Then we may ask the system whether it can perform the rewrite `true => grandparent(X,paul)`.

This theory seems rigorous, general and elegant; many of the difficult problems of multiparadigm programming (including the problem of mutable state) simply do not appear in it. The main criticism to it is that its efficient implementation does not seem easy. For one thing, rewriting should be done *modulo* the ACI rules. Formally speaking, this means that we will have to rewrite not simply terms, but equivalence classes of terms under the ACI relation. Informally this means that a powerful enough matching algorithm should be used so that it can efficiently try all possible permutations of the entities in the configuration (*e.g.* the three entities `(18)`, `(19)` and `(20)` above should be thought of as unordered). For some recent work on AC- and ACI-rewriting see [36, 55, 67].

Another problem is that a rewriting rule should mention and carry to the other side all of the state of an object, even the attributes which do not affect and are not affected by the rule (in `(17)` we have assumed that the only attribute of `Account` is `bal`). This can be avoided using partial matching (with placeholder don't-care variables) inside the structure of the object, *e.g.*

```
(25) rl transfer A from X to Y  < X : Account | bal: XB, RestX >
(26)      < Y : Account | bal: YB, RestY >
(27)   => < X : Account | bal: XB-A, RestX >
(28)      < Y : Account | bal: YB+A, RestY > if XB >= A .
```

(Here the operation `,` inside the object structure should also be ACI. Of course, the addition of don't-cares better be done automatically.) However, this makes the job of the pattern matcher ever harder. There was a similar problem in LOGICAL OBJECTS (Section 6).

Although each individual object is well-structured, the configuration of objects is very much like a blackboard (the same like in LOGICAL OBJECTS) and is to be accessed globally by some interpreter which detects ready-to-interact object-message pairs. The natural way to avoid this huge global space is to allow objects to have subobjects as components (aggregation), which is not provided for in MAUDE.

In conclusion, Rewriting Logic is a marvelous device which integrates in a very simple and natural manner paradigms which were traditionally hard to reconcile: functional, object-oriented, relational and concurrent programming. It "only" remains to implement it efficiently.

9 Conclusions

Of the huge variety of different proposals to accommodate state change in Logic Programming, a couple of approaches stand out as most promising (of course, this selection is highly subjective):

- Traditional but revitalized approaches, as evidenced by Transaction Logic (Section 3). Here the important point is a shift of the emphasis from the controversial non-monotonic issues of elementary updates to the compositional properties of transactions built up from such updates and the possibility to really *program* such transactions.
- Approaches based on Linear Logic (Section 7), whose resource-awareness makes it suitable for reasoning about concurrency and locality, and whose constructive in nature proof search makes it suitable for logic programming. I believe that, despite Andreoli and Pareschi's numerous (and excellent!) works, Linear Logic has not been utilized in full in this area yet.
- Ways to implement Rewriting Logic (Section 8) efficiently, which would make the three major programming paradigms available in a well-integrated framework and would expose unexpected synergism between them.

In any case, it should be concluded that despite numerous efforts, no generally accepted solution to this problem exists yet and there is a large area for research.

References

[1] S. Abiteboul. Updates: A new frontier. In *Second Intl. Conf. on Database Theory*, pages 1–18, 1988.

[2] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Principles of Database Systems (PODS'88)*, pages 240–250. ACM SIGACT-SIGMOD-SIGART, 1988.

[3] S. Abramsky. Computational interpretations of linear logic. *Theoretical Comput. Sci.*, 111:3–57, 1993. Earlier version appeared as Imperial College Technical Report DOC 90/20, Oct. 1990.

[4] V. Alexiev. Applications of linear logic to computation: An overview. Technical Report TR93–18, University of Alberta, Dec. 1993. Submitted to *Bulletin of the IGPL*.

[5] J.-M. Andreoli and R. Pareschi. Logic programming with sequent systems: A linear logic approach. In P. Schroeder-Heister, editor, *Intl. Workshop on Extensions of Logic Programming*, number 475 in LNAI, pages 1–30, Tübingen, Germany, 1989.

[6] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In D. Warren and P. Szeredi, editors, *Intl. Conf. on Logic Programming (ICLP'90)*, pages 495–510, Jerusalem, Israel, June 1990. MIT Press.

[7] J.-M. Andreoli and R. Pareschi. LO and behold! Concurrent Structured Processes. In *ECOOP-OOPSLA'90*, Ottawa, Ontario, 1990. (*SIGPLAN Notices*, 25(10):44–56, Oct. 1990).

[8] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[9] F. Banchilon. A logic programming/object-oriented cocktail. *SIGMOD Record*, 15(3):11–21, Sept. 1986.

[10] A. Bonner and M. Kifer. Transaction logic programming (or, a logic of procedural and declarative knowledge). In *Intl. Conf. on Logic Programming (ICLP'93)*, pages 257–279, Budapest, Hungary, 1993. The full papewr is available as University of Toronto Technical Report CSRI-270, April 1992 (revised 21 May 1993) from `csri.toronto.edu: csri-technical-reports/270/report.ps`.

[11] A. Bonner and M. Kifer. A general logic of state change. Technical report, Computer Systems Research Institute, University of Toronto, 1994. In preparation.

[12] A. Burandt. *Equivalence of Denotational Semantics and Conditions of Standard Models in the Dynamic Logic*. Diploma thesis, University of Karlsruhe, 1991.

[13] W. Chen. Declarative specification and evaluation of database updates. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases (DOOD'91)*, number 566 in LNCS, pages 147–166, Munich, Germany, Dec. 1991.

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

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

[16] J. Conery and S. Haridi. EUDORRA: and object-oriented ANDORRA. Position paper on the ICLP'91 Workshop on OOLP, June 1991.

[17] J. S. Conery. HOOPS: an object-oriented PROLOG. Technical report, University of Oregon, 1987.

[18] J. S. Conery. Object-oriented programming with First-Order Predicate Calculus. Technical Report CIS-TR-87-09, University of Oregon, Aug. 1987.

[19] J. S. Conery. Logical objects. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 420–434, 1988.

[20] A. Davison. POLKA: a PARLOG object-oriented language. Technical report, DOC, Imperial College, London, 1988.

[21] A. Davison. From PARLOG to POLKA in two easy steps. In J. Maluszyński and M. Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming, PLILP'91*, number 528 in LNCS, pages 171–182. Springer-Verlag, 1991.

[22] A. Davison. A survey of logic programming-based object-oriented languages. Technical Report 92/3, University of Melbourne, Jan. 1992. Fourth revision; first published April 1989.

[23] F. Debart, P. Enjalbert, and M. Lescot. Multi-modal logic programming using equational and order-sorted logic. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic Programming (ALP'90)*, number 463 in LNCS, pages 55–69, Nancy, France, Oct. 1990. Springer-Verlag.

[24] 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, Feb. 1988.

[25] R. Fagin, G. Kuper, J. Ullman, and M. Vardi. Updating logical databases. In P. Kanellakis, editor, *Advances in Computing Research*, volume 3, pages 1–18. Plenum Press, 1986.

[26] R. Fagin, J. Ullman, and M. Vardi. On the semantics of updates in databases. In *Principles of Database Systems (PODS'83)*, pages 352–365, Atlanta, GA, Mar. 1983. ACM SIGACT-SIGMOD-SIGART.

[27] L. Fariñas del Cerro. MOLOG: A system that extends PROLOG with modal logic. *New Generation Computing*, 4(1):35–50, 1986.

[28] L. Fariñas del Cerro and A. Herzig. An automated modal logic of elementary changes. In P. Smets, E. Mamdani, D. Dubois, and H. Prade, editors, *Non-Standard Logics for Automated Reasoning*, pages 63–76. Academic Press, 1988.

[29] L. Fariñas del Cerro and A. Herzig. Deterministic modal logic for automated deduction. In L. Aiello, editor, *European Conference on Artificial Intelligence (ECAI'90)*, pages 262–267, Stokholm, Sweden, Aug. 1990.

[30] K. Furukawa, A. Takeuchi, S. Kunifuji, H. Yasukawa, M. Ohki, and K. Ueda. MANDALA: A logic based knowledge programming system. In *International Conference on Fifth Generation Computer Systems*, Tokyo, Nov. 1984.

[31] P.-J. Gailly and J.-L. Binot. The BIM-PROBE experiment. Position paper on the ICLP'91 Workshop on OOLP, June 1991.

[32] J.-Y. Girard. Linear logic. *Theoretical Comput. Sci.*, 50:1–102, 1987.

[33] D. Harel. *First-Order Dynamic Logic*, volume 68 of *LNCS*. Springer-Verlag, 1979.

[34] D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, Oct. 1982.

[35] J. Harland and D. Pym. The uniform proof-theoretic foundation of linear logic programming (extended abstract). In V. Saraswat and K. Ueda, editors, *Intl. Symposium on Logic Programming (SLP'91)*, pages 304–318, 1991. The full paper is available as University of Edinburgh Technical Report ECS-LFCS-90-124, Nov. 1990.

[36] M. Henz. Term rewriting in associative commutative theories with identities. Master's thesis, State University of New York at Stony Brook, Dec. 1991. Available by anonymous FTP from `duck.dfki.uni-sb.de: pub/papers/MT-Henz.ps.Z`.

[37] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic (extended abstract). In *Logic in Computer Science (LICS'91)*, pages 32–42, Amsterdam, July 1991. IEEE Computer Society Press. Full paper to appear in *Journal of Information and Computation* 1992, available from `ftp.cis.upenn.edu: pub/papers/miller/ic92.dvi.Z`.

[38] Y. Ishikawa and M. Tokoro. Concurrent object-oriented knowledge representation language ORIENT84/K: Its features and implementation. In *OOPSLA'86*, Portland, OR, Sept. 1986.

[39] 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, Jan. 1986.

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

[41] R. Jungclaus. *Logic-Based Modeling of Dynamic Object Systems*. PhD thesis, Technical University Braunschweig, Germany, 1993.

[42] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-oriented specification of information systems: The TROLL language. Technical Report Informatik-Bericht 91-04, Technical University Braunschweig, Germany, 1991.

[43] R. Jungclaus, G. Saake, and C. Sernadas. Formal specification of object systems. In S. Abramsky and T. S. E. Maibaum, editors, *International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91), Volume 2: Colloquium on Combining Paradigms for Software Development*, number 494 in LNCS, pages 60–82, Brighton, UK, Apr. 1991. Springer-Verlag.

[44] K. M. Kahn. INTERMISSION—Actors in PROLOG. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*, pages 213–228. Academic Press, 1982.

[45] K. M. Kahn. VULCAN: Logical concurrent objects. In E. S. Shapiro, editor, *Concurrent* PROLOG*: Collected Papers*, volume 2, pages 274–303. MIT Press, 1986.

[46] K. M. Kahn. Objects–a fresh look. In S. Cook, editor, *European Conference on Object-Oriented Programming (ECOOP'89)*, pages 207–223, Nothingham, UK, July 1989.

[47] K. M. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow. Objects in concurrent logic programming languages. In *OOPSLA'86*, Portland, OR, Sept. 1986.

[48] K. M. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow. VULCAN: Logical concurrent objects. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 75–112, Cambridge, MA, 1987. MIT Press.

[49] M. Kanovich. The multiplicative fragment of linear logic is NP-complete. ITLI Prepublication Series X-91-13, University of Amsterdam, 1991.

[50] M. Kanovich. Horn programming in linear logic is NP-complete. In *Logic in Computer Science (LICS'92)*, pages 200–210, Santa Cruz, CA, June 1992. IEEE Computer Society Press. Also University of Amsterdam ITLI Prepublication Series X-91-14.

[51] H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In J. Allen, R. Fikes, and E. Sandewall, editors, *Knowledge Representation and Reasoning (KR'91)*, pages 387–394, Boston, MA, Apr. 1991.

[52] F. Kesim and M. Sergot. On the evolution of objects in a logic programming framework. In ICOT, editor, *Fifth Generation Computer Systems (FGCS'92)*, pages 1052–1060, 1992.

[53] R. kowalski. Database updates in event calculus. *Journal of Logic Programming*, 12(1&2):121–146, Jan. 1991.

[54] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[55] D. Lugiez and J. Moysset. Complement problems and tree automata in AC-like theories. In P. Enjalbert, A. Finkel, and K. Wagner, editors, *Proceedings STACS 93*, volume 665 of *Lecture Notes in Computer Science*, pages 515–524. Springer Verlag, Feb. 1993. Available by anonymous FTP from `duck.dfki.uni-sb.de: pub/ccl/inria-lorraine/stacs93.ps.Z`.

[56] J. Malenfant, G. Lapalme, and J. Vaucher. Coherent state changes for logic programs. Research report LITP 91-01 RXF, Équipe mixte LITP/RXF, Jan. 1991.

[57] S. Manchanda. Declarative expression of deductive database updates. In *Principles of Database Systems (PODS'89)*, pages 93–100. ACM SIGACT-SIGMOD-SIGART, 1989.

[58] S. Manchanda and D. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Logic Programming and Deductive Databases*, Los Altos, CA, 1988. Morgan Kaufmann Publishers.

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

[60] J. Meseguer. A logical theory of concurrent objects. In *ECOOP/OOPSLA'90*, Ottawa, Ontario, 1990. (*SIGPLAN Notices*, 25(10):101–115, Oct. 1990).

[61] J. Meseguer. Rewriting as a unified model of concurrency. In *CONCUR'90: Intl. Conf. on Concurrency Theory*, number 458 in LNCS, pages 384–400, Amsterdam, Aug. 1990. Also Technical Report SRI-CSL-90-02, SRI International, Feb. 1990.

[62] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. Also Technical Report SRI-CSL-91-05, SRI International, Feb. 1991.

[63] J. Meseguer. Multiparadigm logic programming. In *Third Intl. Conf. on Algebraic and Logic Programming*, pages 158–200, Volterra, Italy, Sept. 1992.

[64] J.-J. C. Meyer and R. J. Wieringa. Actor-oriented system specification with dynamic logic. In S. Abramsky and T. S. E. Maibaum, editors, *International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91), Volume 2: Colloquium on Combining Paradigms for Software Development*, number 494 in LNCS, pages 337–357, Brighton, UK, Apr. 1991. Springer-Verlag.

[65] G. Mints. Resolution calculus for the first order linear logic. *Journal of Logic, Language and Information*, (2):59–83, 1993.

[66] V. Pratt. Process logic. In *Principles of Programming Languages (POPL'79)*, pages 93–100, Jan. 1979.

[67] M. Rusinowitch and L. Vigneron. Automated deduction with associative commutative operators. Research Report 1896, Institut National de Recherche en Informatique et Automatique (INRIA), May 1993. Available by anonymous FTP from `duck.dfki.uni-sb.de`: `pub/ccl/inria-lorraine/ AC_deduction.ps.Z`.

[68] D. Sacca, B. Verdonk, and D. Vermeir. Evolution of knowledge bases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology (EDBT'92)*, number 580 in LNCS, pages 230–244, Vienna, Austria, Mar. 1992. Springer-Verlag.

[69] E. Shapiro and A. Takeuchi. Object-oriented programming in CONCURRENT PROLOG. *New Generation Computing*, 1:25–48, 1983.

[70] T. Tammet. Proof strategies in linear logic. Technical Report 70, Programming Methodology Group, Chalmers University of Technology, University of Göteborg, 1993. Accepted to *Journal of Automated Reasoning*. Available from `ftp.cs.chalmers.se`.

[71] T. Uustalu. Combining object-oriented and logic paradigms: A modal logic programming approach. In O. L. Madsen, editor, *European Conference on Object-Oriented Programming (ECOOP'92)*, pages 98–113, June 1992.

[72] D. Warren. Database updates in pure PROLOG. In *Fifth Generation Computer Systems*, pages 244–253. ICOT, 1984.

[73] C. Welsch and G. Barth. Reasoning objects with dynamic knowledge bases. In J.P.Martins and E.M.Morgado, editors, *Fourth Portuguese Conf. on Artificial Intelligence (EPIA'89)*, pages 257–268, Lisbon, Portugal, Sept. 1989.

[74] R. J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Second International Congress on Deductive and Object-Oriented Databases (DOOD'91)*, number 566 in LNCS, pages 431–452, Munich, Germany, Dec. 1991. Springer-Verlag.

[75] M. Winslett. A model based approach to updating databases with incomplete information. *Transactions on Database Systems*, 13(2):167–196, 1988.

[76] M. Winslett. *Updating Logical Databases*, volume 9 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge Univerity Press, 1990.

[77] K. Yoshida and T. Chikayama. A'UM = stream + object + relation. In *OOPSLA'89*, New Orleans, LA, 1989. (*SIGPLAN Notices*, 24(4):55-58, 1989).

[78] K. Yoshida and T. Chikayama. A'UM—a stream-based concurrent object language. *New Generation Computing*, 7:127–157, 1990.

[79] C. Zaniolo. Object-oriented programming in PROLOG. In *International Symposium on Logic Programming*, pages 265–270, Atlantic City, Atlanta, Feb. 1984.