PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Object-Based Versus Logic Programming

Peter Wegner

Brown University, Box 1910, Providence, RI. 02912

pw@cs.brown.edu

Abstract: This position paper argues that mainstream application programming in the 21st century will be object-based rather than logic-based for the following reasons. 1) Object-based programs model application domains more directly than logic programs. 2) Object-based programs have a more flexible program structure than logic programs. 3) Logic programs can be intractable, in part because the satisfiability problem is NP-complete. 4) Soundness limits the granularity of thinking while completeness limits its scope. 5) Inductive, abductive, probabilistic, and nonmonotonic reasoning sacrifice the certainty of deduction for greater heuristic effectiveness. 6) Extensions to deductive logic like nonmonotonic or probabilistic reasoning are better realized in a general computing environment than as extensions to logic programming languages. 7) Object-based systems are open in the sense of being both reactive and extensible, while logic programs are not reactive and have limited extensibility. 8) The don't-know nondeterminism of Prolog precludes reactiveness, while the don't-care nondeterminism of concurrent logic programs makes them nonlogical.

1. Modeling Power and Computability

Object-based programs model application domains more directly than logic programs. Computability is an inadequate measure of modeling capability since all programming languages are equivalent in their computing power. A finer (more discriminating) measure, called "modeling power", is proposed that is closely related to "expressive power", but singles out modeling as the specific form of expressiveness being studied. Features of object-based programming that contribute to its modeling power include:

- assignment and object identity

Objects have an identity that persists when their state changes. Objects with a mutable state capture the dynamically changing properties of real-world objects more directly than mathematical predicates of logic programs.

- data abstraction by information hiding

Objects specify the abstract properties of data by applicable operations without commitment to a data representation. Data abstraction is a more relevant form of abstraction for modeling than logical abstraction.

- messages and communication

Messages model communication among objects more effectively than logic variables. The mathematical behavior of individual objects can be captured by algebras or automoata, but communication and synchronization protocols actually used in practical object-based and concurrent systems have no neat mathematical models.

These features are singled out because they cannot be easily expressed by logic programs. Shapiro [Sh1] defines the comparative expressive power (modeling power) of two languages in terms of the difficulty of mapping programs of one language into the other. Language L1 is said to be more expressive than language L2 if programs of L2 can be easily mapped into those of L1 but the reverse mapping is difficult (according to a complexity metric for language mappings).

The specification of comparative expressive power in terms of mappings between languages is not entirely satisfactory. For example, mapping assembly languages into problem-oriented languages is difficult because of lack of design rather than quality of design. However, when applied to two well-structured language classes like object-based and logic languages this approach does appear promising.

Since logic programs have a procedural interpretation with goal atoms as procedure calls and logic variables as shared communication channels, logic programming can be viewed as a special (reductive) style of procedure-oriented programming. Though language features like nondeterminism, logic variables, and partially instantiated structures are not directly modeled, the basic structure of logic programs is procedural. In contrast, object-oriented programs in Smalltalk or C++ do not have a direct interpretation as logic programs, since objects and classes cannot be easily modeled. Computational objects that describe behavior by collections of operations sharing a hidden state cannot be easily mapped into logic program counterparts.

2. Limitations of Inference and Nondeterministic Control

All deduction follows from the principle that if an element belongs to a set then it belongs to any superset. The Aristotelian syllogism *"All humans are mortal, Socrates is human, therefore Socrates is mortal"* infers that Socrates belongs to the superset of mortals from the fact that he belongs to the subset of humans. This problem can be specified in Prolog as follows:

Prolog clause: *mortal(x) ← human(x).*
Prolog fact: *human(Socrates).*
Prolog goal: *mortal(Socrates).*

The clause *"mortal(x) ← human(x)"*, which specifies that the set of mortals is a superset of the set of humans, allows the goal *"mortal(Socrates)"* to be proved from the fact *"human(Socrates)"*.

A Prolog clause of the form *"P(x) if Q(x)"* asserts that the set of facts or objects satisfying Q is a subset of those satisfying P, being equivalent to the assertion *"For all x, Q(x) implies P(x)"*. A Prolog goal G(x) is true if there are facts in the data-

base that satisfy G by virtue of the set/subset relations implied by the clauses of the Prolog program. Prolog resolution and unification allows the subset of all database facts satisfying G to be found by set/subset inference.

Inferences of the form *"set(x) if subset(x)"* are surprisingly powerful, permitting all of mathematics to be expressed in terms of set theory. But the exclusive use of "set if subset" inference for computation and/or thinking is unduly constraining, since both computation and thinking go beyond mere classification. Thinking includes heuristic mechanisms like generalization and free association that go beyond deduction.

Nondeterminism is another powerful computation mechanism that limits the expressive power of logic programs. Prolog nondeterminically searches the complete goal tree for solutions that satisfy the goal. In a Prolog program with a predicate P appearing in the clause head of N clauses *"P(Ai) → Bi"*, a goal P(A) triggers nondeterministic execution of those bodies Bi for which A unifies with Ai. This execution rule can be specified by a choice statement of the form:

choice (A1|B1, A2|B2, ... , AN|BN) endchoice
nondeterministically execute the bodies Bi of all clauses for which the clause head P(Ai) unifies with the goal P(A).

Bodies Bi are guarded by patterns Ai that must unify with A for Bi to qualify for execution. This form of nondeterminism is called *don't-know nondeterminism* because the programmer need not predict which inference paths lead to successful inference of the goal. Prolog programs explore all alternatives until a successful inference path is found and report failure only if no inference path allows the goal to be inferred.

The order in which nondeterministic alternatives are explored is determined by the system rather than by the user, though the user can influence execution order by the order of listing alternatives. Depth-first search may cause unnecessary nonterminating computation, while breadth-first search avoids this problem but is usually less efficient. Prolog provides mechanisms like the cut which allows search mechanisms to be tampered with. This extra flexibility undermines the logical purity of Prolog programs.

Sequential implementation of don't-know nondeterminism requires backtracking from failed inference paths so that the effects of failed computations become unobservable. Since programs cannot commit to an observable output until a proof is complete, don't-know nondeterminism cannot be used as a computational model for *reactive systems* that respond to external stimuli and produce incremental output [Sh2].

3. Intactability and Satisfiability

Certain well-formulated problems like the halting problem for Turing machines are noncomputable. Practical computability is further restricted by the requirement of tractability. A problem is tractable if its computation time grows no worse than polynomially with its size and intractable if its computation time grows at least exponentially.

The class P of problems computable in polynomial time by a deterministic Turing machine is tractable, while the class NP of problems computable in polynomial time by a nondeterministic Turing machine has solutions checkable in polynomial time though it may take an exponential time to find them [GJ]. The question whether P = NP is open, but the current belief is that NP contains inherently intractable problems, like the *satisfiability problem*, that are not in P.

The satisfiability problem is NP-complete; a polynomial time algorithm for satisfiability would allow all problems in NP to be solved in polynomial time. The fundamental problem of theorem proving, that of finding whether a goal can be satisfied, is therefore intractable unless it turns out that P = NP.

The fact that satisfiability is intractable is not unacceptable especially when compared to the fact that computability is undecidable. But in practice exponential blowup arises more frequently in logic programming than undecidability arises in traditional programming. Sometimes the intractability is inherent in the sense that there is no tractable algorithm that solves the problem. But in many cases more careful analysis can yield a tractable algorithm. Consider for example the sorting problem which can be declaratively specified as the problem of finding an ordered permutation.

sort(x) :- permutation(x), ordered(x).

Direct execution of this specification requires n-factorial steps to sort n elements, while more careful anlysis yields algorithms like quicksort that require only n logn steps. High-level specifications of a problem by logic programs can lead to combinatorially intractable algorithms for problems that are combinatorially tractable when more carefully analyzed.

The complexity of logic problem solving is often combinatorially unacceptable even when problems do have a solution. The intractability of the satisfiability problem causes some problems in artificial intelligence to become intractable when blindly reduced to logic, and provides a practical reason for being cautious in the use of logic for problem solving.

4. Soundness, Completeness and Heuristic Reasoning

Soundness assures the semantic accuracy of inference, requiring all provable assertions to be true, while completeness guarantees inference power, requiring all true assertions to be provable. However, soundness strongly constrains the granularity of thinking, while completeness restricts its semantic scope.

Sound reasoning cannot yield new knowledge; it can only make implicit knowledge explicit. Uncovering implicit knowledge may require creativity, for example when finding whether $P = NP$ or Fermat's last theorem. But such creativity generally requires insights and constructions that go beyond deductive reasoning. The design and construction of software may likewise be viewed as uncovering implicit knowledge by creative processes that transcend deduction. The demonstration that a given solution is correct may be formally specified by "sound" reasoning, but the process of finding the solution is generally not deductive.

Human problem solvers generally make use of heuristics that sacrifice soundness to increase the effectiveness of problem solving. McCarthy suggested supplementing formal systems by a heuristic *advice taker* as early as 1960 [GR], but this idea has not yet been successfully implemented, presumably because the mechanisms of heuristic problem solving are too difficult to automate.

Heuristics that sacrifice soundness to gain inference power include inductive, abductive, and probabilistic forms of reasoning. Induction from a finite set of observations to a general law is central to empirical reasoning but is not deductively sound. Hume's demonstration that induction could not be justified by "pure reason" sent shock waves through nineteenth and twentieth century philosophy.

Abductive explanation of effects by their potential causes is another heuristic that sacrifices soundness to permit plausible

though uncertain conclusions. Choice of the most probable explanation from a set of potential explanations is yet another form of unsound heuristic inference. Inductive, abductive, and probabilistic reasoning have an empirical justification that sacrifices certainty in the interests of common sense.

Completeness limits thinking in a qualitatively different manner from soundness. Completeness constrains reasoning by commitment to a predefined (closed) domain of discourse. The requirement that all true assertions be provable requires a closed notion of truth that was shown by Godel to be inadequate for handling naturally occurring open mathematical domains like that of arithmetic. In guaranteeing the semantic adequacy of a set of axioms and rules of inference, completeness limits their semantic expressiveness, making difficult any extension to capture a richer semantics or refinement to capture more detailed semantic properties. Logic programs cannot easily be extended to handle nonformalized, and possibly nonformalizable, knowledge outside specific formalized domains.

The notion of completeness for theories differs from that for logic; a theory is complete if it is sufficiently strong to determine the truth or falsity of all its primitive assertions. That is, if every ground atom of the theory is either true or false. Theories about observable domains are generally inductive or abductive generalizations from incomplete data that may be logically completed by uncertain assumptions about the truth or falsity of unobserved and as yet unproved ground atoms (facts) in the domain. For example, the *closed-world assumption* [GN] assumes that every fact not provable from the axioms is false. Such premature commitment to the falsity of nonprovable ground assertions may have to be revoked when new facts become known, thereby making reasoning based on the closed-world assumption nonmonotonic.

Nonmonotonic reasoning is a fundamental extension that transforms logic into a more powerful reasoning mechanism. But there is a sense in which nonmonotonic reasoning violates the foundations of logic and may therefore be viewed as nonlogical. The benefits of extending logic to nonmonotonic reasoning must be weighed against the alternative of completely abandoning formal reasoning and adopting more empirical principles of problem solving, like those of object-oriented programming. Attempts to generalize logic to nonmonotonic or heuristic reasoning, while intellectually interesting, may be pragmatically inappropriate as a means of increasing the power of human or computer problem solving. Such extensions to deductive logic are better realized in a general computing environment than as extensions to logic programming languages.

Both complete logics and complete theories require an early commitment to a closed domain of discourse. While the closed-world assumption yields a different form of closedness than that of logical completeness or closed application programs, there is a sense in which these forms of being closed are related. In the next section the term *open system* is examined to characterize this notion as precisely as possible.

5. Open Systems

A system is said to be an *open system* if its behavior can easily be modified and enhanced, either by interaction of the system with the environment or by programmer modification.

1. A *reactive (interactive)* system that can accept input from its environment to modify its behavior is an open system.
2. An *extensible* system whose functionality and/or number of components can be easily extended is an open system.

Our definition includes systems that are reactive or extensible or both, reflecting the fact that a system can be open in many different ways. Extensibility can be *intrinsic* by interactive system evolution or *extrinsic* by programmer modification. Intrinsic extensibility accords better with biological evolution and with human learning and development, but extrinsic extensibility is the more practical approach to software evolution. The following characterization of openness explicitly focuses on this distinction:

1. A system that can extend itself by interaction with its environment is an open system.
2. A system that can be extended by programmer modification (usually because of its modularity) is an open system.

Since extrinsic extensibility is extremely important from the point of view of cost-effective life-cycle management, it is viewed as sufficient to qualify a system as being open. While either one of these properties is sufficient to qualify a system as being open, the most flexible open systems are open in both these senses.

Object-oriented systems are open systems in both the first and second senses. Objects are reactive server modules that accept messages from their environment and return a result. Systems of objects can be statically extended by modifying the behavior of already defined objects or by introducing new objects. Classes facilitate the abstract definition of behavior shared among a collection of objects, while inheritance allows new behavior to be defined incrementally in terms of how it modifies already defined behavior. Classes have the *open/closed* property [Me]; they are open when used by subclasses for behavior extension by inheritance, but are closed when used by objects to execute messages. The idea of *open/closed* subsystems that are both open for clients wishing to extend them and closed for clients wishing to execute them needs to be further explored.

Logic languages exhibiting don't-know nondeterminism are not open in the first sense, while soundness and completeness restrict extensibility in the second sense. To realize reactive openness concurrent logic languages abandon don't-know nondeterminism in favor of *don't-care nondeterminism*, sacrificing logical completeness.

Prolog programs can easily be extended by adding clauses and facts so they may be viewed as open in the second sense. But logical extension is very different from object-based extensibility by modifying and adding objects and classes. Because object-based languages directly model their domain of discourse, object-based extensibility generally reflects incremental extensions that arise in practice more directly than logical extension.

6. Don't-Care Nondeterminism

Don't-care nondeterminism is explicitly used in concurrent languages to provide selective flexibility at entry points to modules. It is also a key implicit control mechanism for realizing selective flexibility in sequential object-based languages. Access to an object with operations *op1, op2, ... , opN* is controlled by an implicit nondetermnistic select statement of the form:

select (op1, op2, ... , opN) endselect

Execution in a sequential object-based system is deterministic from the viewpoint of the system as a whole, but is non-

deterministic from the viewpoint of each object considered as an isolated system. The object does not know which operation will be executed next, and must be prepared to select the next executable operation on the basis of pattern matching with an incoming message. Since no backtracking can occur, the nondeterminism is don't care (committed choice) nondeterminism.

Concurrent porgramming languages like CSP and Ada have explicit don't care nondeterminism realized by guarded commands with guards Gi whose truth causes the associated body Bi to become a candidate for nondeterministic execution:

select (G1‖B1, G2‖B2, ... , GN‖BN) endselect

The keyword *select* is used in place of the keyword *choice* to denote selective don't care nondeterminism, while guards are separated from bodies by ‖ in place of |.

Guarded commands, originally developed by Dijkstra, govern the selection of alternative operations at entry points of concurrently executable tasks. For example, concurrent access to a buffer with an APPEND operation executable when the buffer is not full and a REMOVE operation executable when the buffer is not empty can be specified as follows:

select (notfull‖APPEND, notempty‖REMOVE) endselect

Monitors support unguarded don't-care nondeterminism at the module interface. Selection between APPEND and REMOVE operations of a buffer implemented by a monitor has the following implicit select statement:

select (APPEND, REMOVE) endselect

The monitor operations *wait* and *signal* on internal monitor queues notfull and notempty play the role of guards. Monitors decouple guard conditions from nondeterministic choice, gaining extra flexibility by associating guards with access to resources rather than with module entry.

Consider a concurrent logic program with a predicate P appearing in the head of N clauses of the form *"P(Ai) → Gi‖Bi"*. A goal P(A) triggers nondeterministic execution of those bodies Bi for which A unifies with Ai and the guards Gi are satisfied. This execution rule can be specified by a select statement of the form:

select ((A1;G1)‖B1, (A2;G2)‖B2, ... , (AN;GN)‖BN) endselect
Bi is a candidate for execution if A unifies with Ai and Gi is satisfied

Since no backtracking can occur once execution has committed to a particular select alternative, the nodeterminism is don't-care nondeterminism. However, don't care nondeterminism in concurrent logic languages is less flexible than in object-based languages because data abstraction and object-based message communication is not supported.

Don't-care nondeterminism is useful in realizing reactive flexibility, but is neither necessary nor sufficient for concurrent systems. Concurrent nonreactive systems for very fast computations are commonplace, while sequential object-based systems are reactive but not nonconcurrent. Reactiveness and concurrency are orthogonal properties of computing systems. Don't-care nondeterminism is primarily concerned with enhancing reactive flexibility and is not strictly necessary for concurrency.

Nondeterministic selection is relatively complex because it combines merging of incoming messages from multiple sources with selection among alternative next actions by pattern matching. The essential nondeterminism in concurrent systems arises from uncertainty about the arrival order (or processing order) of incoming messages and is modeled by implicit nondeterministic merging of streams rather than by explicit selection. For example, the nondeterministic behavior of a bank account with $100.00 when two clients each attempt to withdraw $75.00 depends not on selective don't-care nondeterminism but simply on the arrival order of messages from clients.

7. Are Concurrent Logic Programs Nonlogical?

Don't-care nondeterminism serves to realize reactive computations and also to keep the number of nondeterministic alternatives explored to a manageable size. But it may cause premature commitment to an inference path not containing a solution at the expense of paths that possibly contain solutions. Don't-care nondeterminism is nonmonotonic since adding a rule may have the effect of preventing commitment to an already existing rule. Logic programs employing don't-care nondeterminism are *incomplete* in the sense that they may fail to prove *true* assertions that would have been derivable by don't-know nondeterminism from the same set of clauses. It becomes the responsibility of the programmer to make sure that programs do not yield different results for different orders of don't-care commitment.

Under don't-care nondeterminism the result of a computation from a set of clauses depends on the order of don't-care commitment. This weakens the claim that concurrent logic languages are logical, reducing them to the status of ordinary programming languages. Clauses lose the status of inference rules, becoming mere computation rules. As hinted at in [Co], don't care nondeterminism takes the L out of LP, reducing logic programming to programming. The committed-choice inference paradigm loses the status of a proof technique and becomes a computational heuristic whose rules impose a rigid structure on both conceptualization and computation.

Don't-know nondeterminism provides a computational model for logical inference, while don't-care nondeterminism models incremental, reactive computation, but sacrifices logical inference. Reactive systems are open systems in the sense that they may react to stimuli from the environment by returning results and changing their internal state. Objects are a prime example of reactive systems, responding interactively to messages they receive. The inability of don't-know nondeterminism to handle reactiveness is a serious weakness of both logic programs and deductive reasoning. The fundamental reason for this is the inability of inference systems to commit themselves to incremental output.

While pure logic programming is incompatible with reactiveness it is definitely compatible with concurrency. The components of logical expressions may be concurrently evaluated. Universal and existential quantification, which is simply transfinite conjunction and disjunction, can be approximated by concurrent evaluation of components. Reactiveness is orthogonal to concurrency in the sense that concurrent nonreactive systems for very fast computations are commonplace, while sequential object-based systems are reactive but not nonconcurrent. However, reactive responsiveness is as important in large applications as concurrency. The identification of reactiveness and concurrency as independent goals of system design marks a step forward in our understanding of system requirements.

The *process interpretation* of concurrent logic programs views goal atoms as processes and logic variables as streams.

The set of goals at any given point in the computation becomes a dynamic network of processes that may be reconfigured during every goal-reduction step. Every concurrent logic program has a process interpretation, but concurrent object-oriented application programs cannot be directly mapped into concurrent logic programs. Thus concurrent logic programs are less expressive than object-oriented programs in the sense of [Sh1]. Logical processes have no local state; they are atomic predicates whose granularity cannot be adapted to the granularity of objects in the application domain. Concurrent logic programs give up their claim to be logical without gaining the communication and computation flexibility of traditional concurrent languages.

8. Are Multiparadigm Logic/Object Systems Possible?

Can the object-based and logic programming paradigms be combined to capture both the decomposition and abstraction power of objects and the reasoning power of logic? Experience suggests that logic is not by itself a sufficient mechanism for problem solving and that combining logical and nonlogical paradigms of problem solving is far harder than one might expect. Logic plays a greater role in verifying the correctness of programs than in their development and evolution. Finding a solution to a problem is less tractable than verifying the correctness or adequacy of an already given solution. For example, solutions of problems in NP can be verified in polynomial time but appear to require exponential time to find. Verification and validation is generally performed separately after a program (or physical engineering structure) has been constructed.

The logic and object paradigms have different conceptual and computational models. Logic programs have a clausal inference structure for reasoning about facts in a database, while object-based programs compute by message passing among heterogeneous, loosely-coupled software components. Logical reasoning is top-down (from goals to subgoals), while object-based design is bottom-up (from objects of the domain). Object-based programs lend themselves to development and evolution by incremental program changes that directly correspond to incremental changes of the modeled world. Inference rules provide less scope for incremental descriptive evolution, since rules for reasoning are not as amenable to change as object descriptions.

ICOT's choice of logic programming as the vehicle for future computing contrasts with the US Department of Defense's choice of Ada. Because Ada was designed in the 1970s, when the technology of concurrent and distributed software components was still in a primitive state, it has design flaws in its module architecture. But its goals are squarely in the object-oriented tradition of model building based on abstraction.

During the past 15 years we have accumulated much experience in designing object-oriented, distributed, and knowledge-based systems. The international computing community may well be ready for a major attempt to synthesize this experience in developing a standard architecture for distributed, intelligent problem solving in the 21st century. Such an architecture would be closer to the object-oriented than to the logic programming tradition.

Next-generation computing architectures should try to synthesize the logic and object-oriented traditions, creating a multiparadigm environment to support the cooperative use of both abstraction and inference paradigms. For example, an object's operations could in principle be implemented as logic programs, though the use of Prolog as an implementation language for object interfaces presents some technological problems. Perhaps technological progress in the 21st century will resolve these problems so that multiparadigm environments can be developed facilitating the cooperative application of both abstraction and inference paradigms.

Problem solving is a social process that involves cooperation among people, especially for large projects with a long life cycle. Decomposition of a problem into object abstractions is important both for cooperative software development and for incremental maintenance and enhancement. While object-oriented problem representation is not uniformly optimal for all problems, it does provide a robust framework for cooperative incremental software evolution for a much larger class of problems than logical representation.

The early optimism that artificial intelligence could be realized by a *general problem solver* gave way in the 1960s to an appreciation of the importance of domain-dependent knowledge representation. The debate concerning declarative versus procedural knowledge representation was resolved in the 1970s in favor of predicate calculus declarative representation. AI textbooks of the 1980s [CM, GN] advocate the predicate calculus as a universal framework for knowledge representation, with domain-dependent behavior modeled by nonlogical predicate symbols satisfying nonlogical axioms.

The logic and network approaches to AI have competed for research funds since the 1950s [Gr], with the logic-based symbol system hypothesis dominating in the 1960s and 1970s and distributed pattern matching and connectionist learning networks staging a comeback in the late 1980s [RM]. The idea that intelligence evolves through learning is an appealing alternative to the view that intelligence is determined by logic, but attempts to realize nontrivial intelligence by learning have proved combinatorially intractable. Distributed artificial intelligence research [BG] and Minsky's *The Society of Mind* [Min], view problem solving as a cooperative activity among distributed agents very much in the spirit of object-oriented programming. Ascribing mental qualities like beliefs, intentions, and consciousness to agents is likewise compatible with the object-oriented approach.

9. References

[BG] A. H. Bond and L. Gasser, *Readings in Distributed Artificial Intelligence*, Morgan-Kaufman 1988.

[Co] J. Cohen, Introductory remarks for the special CACM issue on logic programming, CACM, March 1992

[CM] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*, Addison-Wesley 1984

[GJ] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman 1978.

[GN] M. R. Genesereth and N. J. Nilson, *Logical Foundations of Artificial Intelligence*, Morgan-Kaufmann 1987.

[Gr] *The Artificial Intelligence Debate*, Editor Stephen Graubard, MIT Press 1988

[Me] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall International 1988.

[Min] Marvin Minsky, *The Society of Mind*, Simon and Schuster, 1987

[RM] D. E. Rumelhart and J. L. McLelland, *Parallel Distributed Processes*, MIT Press 1986.

[Sh1] E. Shapiro, Separating Concurrent Languages with Categories of Language Embeddings, TR CS91-05, Weizmann Institute, March 1991

[Sh2] E. Shapiro, The Family of Concurrent Programming Languages, *Computing Surveys*, September 1989