MODELING SIMULTANEOUS EVENTS WITH DEFAULT REASONING AND TIGHT DERIVATIONS

ALLEN VAN GELDER

▷ This report describes how an arbitrator for the game of Diplomacy was implemented in PROLOG, and discusses the advantages of logic programming for this type of problem. The design of the program is greatly simplified by using a default reasoning style to express many of the game rules. The technique of default reasoning is to specify a predicate (or property or situation) by stating sufficient conditions for a general rule, and then stating exceptions to it. Logic programming supports the expression of default reasoning quite directly with negation as failure. Tight derivations were employed to analyze cycles correctly. A tight derivation is one in which no goal has an identical ancestor. Speaking informally, any provable goal is provable by a tight derivation, so restricting the proof searches to tight derivations does not lose any proofs. Only the parts of the program that are susceptible to infinite recursions check for tightness. It is known that the use of tight derivations corresponds to a certain "preferred", or "canonical", minimal model for many programs that contain negation as failure; in such programs, there is normally no *minimum* model. Care must be taken in the way defaults are expressed to achieve proper correspondence between the model and the game rules. ◁

1. INTRODUCTION

This report describes how an arbitrator for the game of Diplomacy was implemented in PROLOG, and discusses the advantages of logic programming for this type of problem. This problem is a member of a large class of problems that can be formulated as discrete systems whose changes of state are governed by a set of "control" operators. Automated process control systems and multiple robot systems are examples of such discrete systems. A characteristic of large "real world" systems

Address correspondence to Allen Van Gelder, Computer and Information Sciences, Room 225AS, University of California, Santa Cruz, CA 95064 (avgsaturn.ucsc.edu).

Received May 1987; accepted May 1988.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/90/$3.50

that is often absent from "toy" systems (but present in Diplomacy) is that many control operators can be applied simultaneously. Typically, different operators mainly affect different parts of the system, but have some degree of interaction that prevents them from being treated as independent.

1.1. Discrete Systems

Discrete systems for our purposes are systems that change only at discrete points in time and are described by a finite or countably infinite set of parameters. Parameter *values* may have a continuous range, such as real numbers, in general, but we shall limit discussion to finite systems for the most part. Frequently in artificial intelligence applications an underlying continuous system is abstractly represented by a finite discrete system as an approximation.

Many models of discrete systems are based on the concepts of a *state*, *operators*, and rules that govern *changes in state*. To be computationally tractable, the rules should be local in nature. The *state* is frequently represented (actually or conceptually) as a large collection of facts. Rules for change are easiest to understand and apply if they involve testing and changing relatively few elements of the state. Complex problems frequently arise if many operators are applied simultaneously.

Issues of simultaneous operations also frequently arise in database update situations. Databases are in a consistent state when they satisfy certain constraints. An update transaction may require changes to many records, or facts. After all changes, the database should again be in a consistent state; however, if the changes are done serially, intermediate inconsistent states may be created. The problem is how to treat the required changes properly as simultaneous operations.

Another view of Diplomacy arbitration is that the rules are a body of law to be applied to a case, the facts of which are represented by the players' orders. The idea of applying logic programming to the interpretation of law has been explored by Sergot et al [14]. As is frequently the case with laws, the rules of Diplomacy have evolved since the game was introduced by Games Research, Inc. in 1961; the original rules were stated in 8 pages, and by 1982 had expanded to 11 pages.

1.2. The Game of Diplomacy

Diplomacy is a board game in which seven players, representing "great powers" of Europe in the early 1900s, manage their military units by simultaneously giving orders for all of their units. Each unit (a fleet or an army) may be ordered to move, hold, or assist another unit. Players can see the current positions of all units and can negotiate with each other before deciding upon their orders, whence the game's name. Orders are written privately, then opened all at once. The rules of the game are applied to arbitrate conflicts and determine which orders succeeded. Successful moves are then carried out on the board, and another round of play commences. Although there are many other elements to the game, such as retreating, disbanding, and building new units, in this paper we shall concentrate on the problem of arbitrating the player's orders in accordance with the published rules of the game. Avalon Hill Game Co., which currently markets Diplomacy, publishes a booklet containing these rules [3] and includes it with the game equipment.[1] In later sections

[1]Avalon Hill has requested the author to state that the company has no association with the author or this article.

we introduce enough of the rules to allow a reader who is unfamiliar with the game to follow the discussion of programming issues involved.

1.3. The Implementation of an Arbitrator

Diplomacy arbitration was chosen as a vehicle for investigation because it has sufficient complexity to bring issues to light, yet is well defined and compact enough to be manageable by one person. Representation as a discrete system with operators is straightforward: The positions and owners of the military units constitute the state, and the orders represent the operators.

Other concurrent efforts to implement a Diplomacy arbitrator in other languages offered an opportunity for comparison of approaches. (The other efforts are not finished at this writing.) In another research project at Stanford, Diplomacy is being used to study problems of cooperation and commitment among independent autonomous agents whose goals are partially conflicting and partially consistent [6].

Logic programming has also been used to implement expert systems for problems in the game of bridge [13] and other games.

The program discussed here is written in CPROLOG in a portable style that is easily converted to DEC-20 PROLOG and compatible commercial versions. It represents about 100 hours of work. An initial version was put into "operation" after about four weeks. (By this we mean it was given to others to use.) A substantial revision of the initial version was undertaken when the author obtained a copy of the up-to-date rules. (The author is indebted to Kevin Knight of Carnegie Mellon University for coding the database statements that represent the map, for providing and reviewing many program tests, for advising the author that his idea of the rules was out of date.) The current program comprises about 1000 lines, including comments and a reasonable user interface. It has been in bug-free use for about six months as of this writing.

A fair amount of the effort went into making the program usable by someone with virtually no knowledge of PROLOG. PROLOG's operator declaration feature was used to allow orders to be written in a natural style close to the way players normally write their orders. A parser that "runs both ways" converts orders between the external and the internal representation. The parser inserts uninstantiated variables when certain parts of an order are elided, and fills them in as more information becomes available. For example, if two of the orders are:

```
fra a bel s nth ⇒ hol.
eng f nth ⇒ hol.
```

then the first order, when written out after the analysis, appears as

```
fra a bel s eng f nth ⇒ hol.
```

which is read by Diplomacy players as "French army in Belgium supports English fleet in North Sea into Holland." (Players actually use this terse style for writing orders; standard abbreviations are suggested in the rule booklet.)

After loading the program into CPROLOG (which has no complier), the user types

`ro` ⟨*input file name*⟩.

which causes the orders to be read in and checked for errors. If satisfied, the user

then types

`res` ⟨*output file name*⟩.

which causes the orders to be arbitrated, and the results written on the named output file. Other options are displayed by typing "`h`". The commands can be repeated to process other sets of orders, to reread orders after they have been corrected, etc. Each set of orders is processed in a few seconds on a Digital Equipment VAX 780.

2. MODELING DISCRETE SYSTEMS

Logic programming is a natural vehicle for simulation of a discrete system in which the effects of operators are described by rules. This is especially true when the system is largely nonnumeric. In many cases the operator has an effect on only one or a few state elements. In a purely logical specification an additional formalism is needed to specify that the operator does *not* affect other state elements. One approach is to use *frame axioms* [10]. However, these prove to be computationally expensive, and do not generalize naturally to simultaneous application of several operators.

Nonmonotonic reasoning encompasses a class of alternatives to frame axioms that are the subject of much current research. Two similar techniques in this class are the *closed world assumption* [12] and *circumscription* [8, 9]. Informally, the closed world assumption states that, with various qualifications, if a fact cannot be proven true, then it may be assumed to be false; that is, certain facts are *false by default*. This idea is closely connected to and generalized by circumscription. The circumscription axioms generally state that if a model, represented as a set of positive facts (or points where predicates are true), has a proper subset that also is a model, then the larger model should be rejected. The intuition is that the rejected model contains "unnecessary" facts. One might say that the closed world assumption and circumscription are biased against positive facts. This bias reflects the experience that normally in the real world most things are not true, and if they are, there is a reason.

2.1. Axiomatization of the Problem

The first step in developing a program to simulate a discrete system is to describe the system and its behavior by axioms Normally, these axioms fall into several distinct categories:

Facts that are true in all states.

Constraints on states.

Rules for operators.

These axioms are static. The state of the system at a particular time can usually be represented as a set of facts. The state can change for three reasons:

Intrinsic operators, often called *equations of motion* in continuous systems.

Control operators, applied by some agent(s) for a purpose.

Random operators.

The reasons for change of state can interact, and not all three kinds need be present in the system. Perhaps the fundamental problem for a discrete system is the forward (or backward) *time projection problem*:

> Given a set of simultaneously applied operators and the current state, what is the successor (or predecessor) state?

Diplomacy arbitration requires the solution of a forward time projection problem, with the players' orders filling the role of operators.

2.2. *Elementary Axioms of Diplomacy*

The static axioms are derived by translation of the rules of the game. A complete description of the rules would be inappropriate, but we give enough details to illustrate the problems involved in automating them. The axioms that express the "eternal truths" are simply facts:

> the names of spaces on the board, and whether each is land or sea;
>
> which spaces are adjacent for army movement, which for fleet movement;
>
> which spaces may be occupied by armies (land), which by fleets (sea, and land adjacent to sea);
>
> the names of the seven "great powers", and other miscellany.

The constraints of "legal" states are also quite simple to state informally:

> Two units may not occupy the same space.
>
> Each unit must occupy exactly one space.

3. COMPLETE AXIOMATIZATION IS IMPRACTICAL

By far the most complicated part of the axiomatization involves the orders. In fact, formulating these axioms overlaps writing the program to a large extent. This overlap is one of the reasons that this problem is particularly amenable to logic programming.

To illustrate the complications that can arise in arbitration, we show a typical (partial) game situation in Figure 1, and refer to it throughout the following discussion. For brevity, we name units by the spaces they occupy, saying "*Rumania*" instead of "the army in *Rumania*", etc.

If two pieces attempt to occupy the same space, either both moving or one moving and one holding (remaining in place), the conflict is resolved by determining which order has the greater "strength". The order's strength is measured by counting its supports. A move order succeeds only if it has greater strength than any conflicting order. If a move order is unsuccessful it effectively becomes a hold order. This basic idea has many exceptions and qualifications, however. For example, in Figure 1 *Sevastopol* has greater strength than *Rumania*, but even if *Rumania* fails to move (converting its order to "hold"), *Sevastopol* does not "dislodge" it because both units belong to the same country.

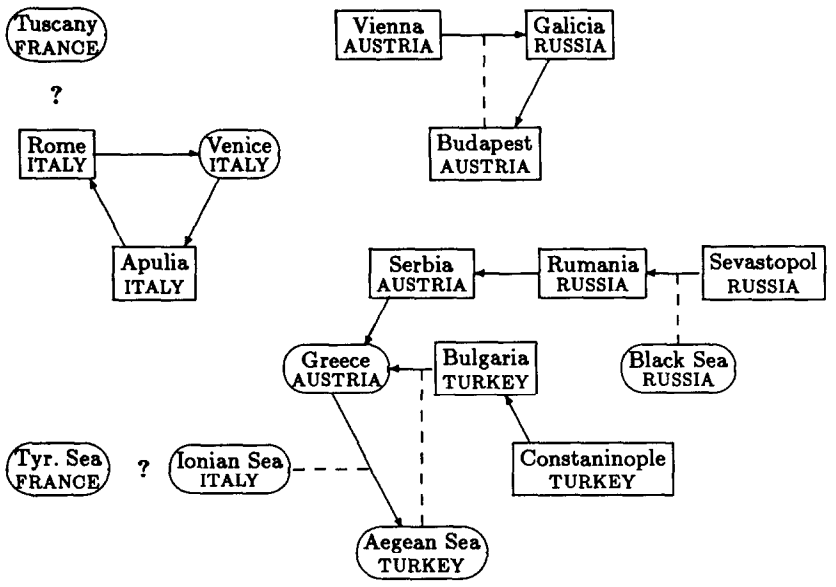

FIGURE 1. Typical simultaneous orders to be arbitrated. Ovals are fleets, and rectangles are armies. Dashed lines represent *support* orders.

3.1. Effective Use of Defaults and Exceptions

The paramount qualification on support orders' effectiveness is that a support order is nullified (cut) if the supporting unit is attacked. Thus, if *Tyr. Sea* is ordered to move to the *Ionian Sea*, then *Ionian Sea* is said to be attacked, and its support for the *Greece* move is "cut", even though the attack on *Ionian Sea* does not succeed. But this qualification has an exception (several, in fact). If the attack is from the space into which support is given, as is the case in which *Galicia* is attacking *Budapest*, this attack does not nullify the support. However, even this exception has an exception!

The exception to the exception occurs in the *Aegean Sea*. Assume *Tyr. Sea* does not attack *Ionian Sea*. By the above exception, the *Greece* attack does not nullify the support of the *Aegean Sea*, because the support is for an attack on *Greece* itself. But Greece is attacking *Aegean Sea* with strength two, and *Aegean Sea*, having no supports, is holding with strength only one. *Greece* prevails and *Aegean Sea* is said to be *dislodged*. In this event the exception to the exception applies, and the support of the *Aegean Sea* is nullified.

Many of the game rules are expressed in this manner. A general rule is stated, then exceptions, exceptions to the exceptions, and so on, qualify it. This naturally leads to a formalization that uses defaults.

3.2. Recursive Specifications are Needed

The nontrivial aspect of arbitration is that deciding whether one move is successful may require knowing whether a different move is successful, leading to a recursive computation. In the example, *Constantinople* succeeds into *Bulgaria* only if *Bulgaria*

moves. *Bulgaria* succeeds into *Greece* only if it has greater strength than *Serbia*, who is also attempting to move to *Greece*. For *Bulgaria* to have greater strength, the support of *Aegean Sea* must not be nullified. As discussed above, the support of *Aegean Sea* is nullified if and only if the *Greece* move to *Aegean Sea* is successful, and this depends on whether *Tyr. Sea* attacks *Ionian Sea*.

It is also possible for a set of moves to form a cycle, as is the case for *Rome*, *Venice*, and *Apulia*. In a cycle such that no move can dislodge the unit in its path, they may all succeed. However, if a unit outside the cycle attempts to move into it, such as *Tuscany* to *Rome*, that may block one move in the cycle, and consequently block the entire cycle. Correct handling of cycles has proved to be a stumbling block in other implementation attempts.

4. DEFAULT REASONING AND NEGATION AS FAILURE

The principal mechanism in logic programming that supports specification in terms of defaults and exceptions is *negation as failure*. The fundamental ideas were introduced by Clark in a logic programming context [4], and by Reiter in a database context [12]. PROLOG supports a direct expression of this concept with a "not provable" operator, $\backslash +$ in many dialects. (Observe that $\backslash +$ is approximately a linear transformation of $\nvdash$.) The straightforward way to express in PROLOG that a support order is "O.K." unless it is found to be "cut", based on the previously discussed rules, is:

$$\begin{aligned} supportOK(L, A, B) &\leftarrow supportOrdered(L, A, B), \\ &\quad \backslash + supportCut(L, A, B). \\ supportCut(L, A, B) &\leftarrow moveOrdered(X, L), \\ &\quad \backslash + sameCountry(X, L), \\ &\quad \backslash + X = B. \\ supportCut(L, A, B) &\leftarrow moveOrdered(B, L), \\ &\quad moveOK(B, L). \end{aligned}$$

We have followed a general technique in arriving at the above rules:

1. Find necessary conditions for the predicate to be "true by default".
2. Define one or more new predicates to cover the exceptions, and write their rules.
3. The original predicate is true when the necessary conditions are true and the applicable exception predicate is not provable.

Notice that step 2 may recursively use the default-exception paradigm.

In many cases the predicates will have a natural hierarchy based on which predicates are defined in terms of which. Programs in which negative dependencies among predicates are acyclic are called *stratified*, or *free of recursive negation* [1, 7, 11, 15]. Such programs have a natural model that is based on the dependence structure of the predicates. When the domain is finite and the rules are not recursive, this model agrees well with PROLOG's operational semantics. As soon as recursive rules appear, there is the danger that PROLOG will not terminate on a

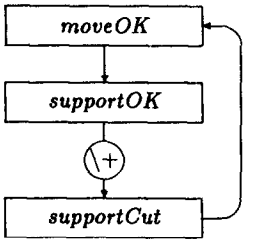

FIGURE 2. A negative cycle of predicates.

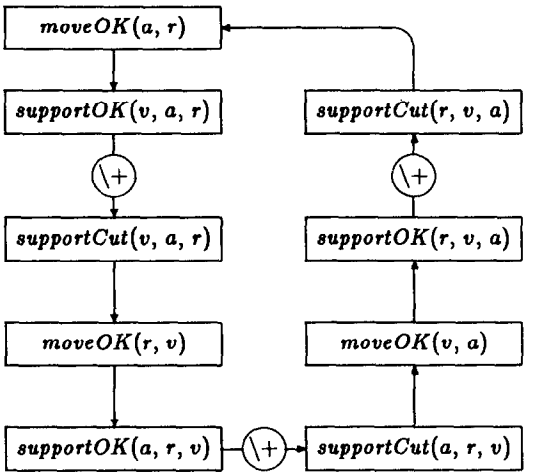

FIGURE 3. A potential negative cycle of atoms. The symbols a, r, and v abbreviate Apulia, Rome, and Venice.

proof attempt. In many other cases the problem will exhibit cycles in the negative dependencies, introducing additional problems.

In fact, Diplomacy rules lead to recursive rules that are not stratified. It is clear from the earlier discussions that a predicate *moveOK* will depend on a predicate *supportOK*. Examination of the PROLOG rules given above shows that *supportOK* depends negatively on *supportCut*, which in turn might depend on *moveOK*. Thus a negative cycle of length three exists at the granularity of predicates, as shown in Figure 2. The situation is more complicated at the granulartiy of atoms, but negative cycles of length 9 can be constructed, as shown in Figure 3.

Attaching a formal meaning to logic programs with potential negative cycles even at the granulartiy of atoms is a research topic that is beyond the scope of this report. Definitions of "unique stable" and "well-founded" models have been proposed to cover programs, such as this Diplomacy arbitrator, in which potential negative cycles can be constructed, but are really immaterial because other subgoals in the rules needed to construct the cycle are unsatisfiable [5, 16]. The important observation for this program is that the potential negative cycle in Figure 3, and others like it, is pertinent only if some unit has two orders, an impossibility in a legal set of Diplomacy orders. Therefore the fact that this program checks the orders for such errors before attempting to arbitrate them is actually necessary to assure its termination.

5. SIMULTANEOUS OPERATIONS AND CYCLES

For most systems, the effect of operators that are applied simultaneously cannot be achieved by applying each operator in succession. In Diplomacy, each of two move orders might lead to a consistent new state if applied alone. But if both are applied, the units might occupy the same space in the new state, violating a constraint axiom. The bulk of the rules deal with resolution of conflicting orders. Applying two conflicting orders in sequence frequently gives the wrong result no matter what sequence is chosen.

Essentially, move orders either succeed or fail. If a move fails, the unit continues to occupy its original space, as though it had been ordered to hold, but it may be *dislodged*. If a move is ordered into an occupied space, then either the move fails or the original unit is dislodged, to preserve the constraint on the new state. The rules for deciding these conflicts are complicated and involve counting up support orders on both sides. We shall address only one aspect of this problem, one that involves default reasoning in an important way.

The crux of the problem is that a *set* of move orders that form a cycle must all succeed or all fail, when no unit has sufficient strength to dislodge the unit in its path. Thus we might formulate rules:

$$\begin{aligned} \textit{move OK}(A, B) \leftarrow\ & \textit{moveOrdered}(A, B), \\ & \textit{strongest}(A, B), \\ & \backslash + \textit{occupied}(B). \end{aligned}$$

$$\begin{aligned} \textit{moveOK}(A, B) \leftarrow\ & \textit{moveOrdered}(A, B), \\ & \textit{strongest}(A, B), \\ & \textit{occupied}(B), \\ & \textit{dislodges}(A, B). \end{aligned}$$

$$\begin{aligned} \textit{moveOK}(A, B) \leftarrow\ & \textit{moveOrdered}(A, B), \\ & \textit{strongest}(A, B), \\ & \textit{occupied}(B), \\ & \textit{moveOK}(B, X). \end{aligned}$$

These rules are oversimplified, ignoring many Diplomacy cases, to emphasize the important features of handling cycles. The predicate *strongest* pertains to units attempting to move to B, while *dislodges* pertains to the unit already occupying B. As an operational PROLOG procedure we can see an immediate problem that it can go into a recursion loop if the move orders form a cycle.

However, there is a deeper problem. Assuming that the orders are such that either all moves succeed or all fail, in the minimum model, *moveOK* is false for all moves in the loop. Thus even if some loop checking mechanism is added to PROLOG, the above rules will produce the incorrect answer, for the rules of Diplomacy specify that moves around a cycle of three or more succeed.

The solution is to employ the default-exceptions paradigm, and to use tight derivations to minimize the exceptions. A *tight derivation* is one in which no goal node is identical to any of its proper ancestors. It is easy to show (see [15]) that any provable goal can be proven with a tight derivation, so restricting the proof search

to such derivations will not cause a provable goal to fail. In the general case this is easier said than done: Because of variables in the goals, it is unpredictable whether certain ancestors will be identical when they are eventually instantiated. However, in PROLOG rules where the recursive goal will be variable-free by the time it becomes an ancestor, it is easy to maintain a stack of ancestors as an additional argument, and thereby detect repetition of ancestors. By failing any proof attempt that repeats ancestors on a path, we admit only tight derivations.

The required insight for the move cycle situation is that a move succeeds by default, unless an exception applies. Let *moveOK* represent the success of a move, and let *moveFails* describe the exceptions. We see that the recursion is now on the exception predicate:

$$moveOK(A, B) \leftarrow moveOrdered(A, B),$$
$$\backslash + moveFails(A, B).$$
$$moveFails(A, B) \leftarrow \backslash + strongest(B).$$
$$moveFails(A, B) \leftarrow occupied(B),$$
$$\backslash + dislodges(A, B),$$
$$\backslash + moveOrdered(B, X).$$
$$moveFails(A, B) \leftarrow occupied(B),$$
$$\backslash + dislodges(A, B),$$
$$moveOrdered(B, X),$$
$$moveFails(B, X).$$

These rules have the desired semantic properties in that the "canonical" model gives the correct answer (for the oversimplified game rules), but they will not execute correctly in PROLOG.

Since existing PROLOG implementations do not check ancestors automatically, the above rules need to be modified to include their own ancestor checking. We use syntax in which [ ] is the empty list and [H | T] denotes a list with head H and tail T:

$$moveOK(A, B) \leftarrow moveOrdered(A, B),$$
$$\backslash + moveFails(A, B, [\,]).$$
$$moveFails(A, B, Ancs) \leftarrow \backslash + strongest(B).$$
$$moveFails(A, B, Ancs) \leftarrow occupied(B),$$
$$\backslash + dislodges(A, B),$$
$$\backslash + moveOrdered(B, X).$$
$$moveFails(A, B, Ancs) \leftarrow occupied(B),$$
$$\backslash + dislodges(A, B),$$
$$moveOrdered(B, X),$$
$$\backslash + member(A, Ancs),$$
$$moveFails(B, X, [A|Ancs]).$$

We remark that, while sufficient to ensure termination and correctness under these circumstances (in which the ancestor chain is variable-free and the domain is finite), simple ancestor checking is not necessarily efficient. Consider the move orders as defining a directed graph. In our application the graph has outdegree one. Consequently, the number of simple paths, and hence the running time, is polynomial. However, for general graphs, a program that uses ancestor chains to detect cycles can use exponential time, as it explores all simple paths. In such applications some form of node marking or dynamic programming [2] is recommended. In the artificial intelligence literature, dynamic programming techniques are frequently called "memo-ization".

6. CONCLUSION

The principal gain in our formulation of defaults was that we were able to reduce the problem of deciding whether a set of facts were all "true" (a whole cycle of moves) to the problem of deciding whether single facts were "true" (a move fails). Deciding single facts is often straightforward with a logic program. The important techniques were negation as failure and ancestor checking, when needed. We believe this is a promising approach for a variety of problems.

REFERENCES

1. Apt, K. R., Blair, H., and Walker, A., Towards a Theory of Declarative Knowledge, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988.
2. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., Ullman, *Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
3. *Rules for Diplomacy*, 2nd ed., Avalon Hill Game Co., 4517 Harford Rd., Baltimore, MD, Feb. 1982.
4. Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum, New York, 1978, pp. 293–322.
5. Gelfond, M. and Lifschitz, V., The Stable Model Semantics for Logic Programming, in: *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, WA, 1988, pp. 1070–1080.
6. Ginsberg, M. and Young, R. M., A Formal Language for Negotiation and Commitment, Computer Science Dept., Stanford University, 1988.
7. Lifschitz, V., On the Declarative Semantics of Logic Programs with Negation, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988.
8. McCarthy, J., Circumscription—a Form of Non-monotonic Reasoning, *Artificial Intelligence* 13(1):27–39 (1980).
9. McCarthy, J., Applications of Circumscription to Formalizing Common Sense Knowledge, in: *AAAI Workshop on Non-monotonic Reasoning*, 1984, pp. 295–323.
10. Nilsson, N., *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.
11. Przymusinski, T., On the Declarative Semantics of Deductive Databases and Logic Programs, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988.
12. Reiter, R., On Closed World Databases, in: H. Gallaire and J. Minker (eds)., *Logic and Databases*, Plenum, New York, 1978, pp. 55–76.

13. Sterling, L., and Nygate, Y., PYTHON: An Expert Squeezer, J. *Logic Programming*, this issue.
14. Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P., and Cory, H. T., The British Nationality Act as a Logic Program, *Comm. ACM* 29(5):370–387 (1986).
15. Van Gelder, A., Negation as Failure Using Tight Derivations for General Logic Programs, *J. Logic Programming* 6(1): 109–133 (1989).
16. Van Gelder, A., Ross, K. A., and Schlipf, J. S., Unfounded Sets and Well-Founded Semantics for General Logic Programs, in: *Proceedings of the Seventh ACM Conference on Principles of Database Systems*, Austin, TX, 1988, pp. 221–230.