Counterfactual exceptions in deductive database queries

Troels Andreasen and **Henning Christiansen**[1]

Abstract. This paper suggests a new construct to capture negative hypothesis in database query languages. Counterfactual exceptions, as the construct is called, are specialized constraints in queries, that serve as means to suppress part of the database. The expressibility obtained is closely related to what is captured by possible world counterfactuals, but the semantic characterization becomes simpler and an implementation can be obtained in a straightforward way. The logical semantics is described in terms of model and completion constructions. An inference system is obtained by a modification of *modus ponens*. Also a generalization into a language with hypothetical implication goals and positive as well as negative hypotheses is suggested.

1 INTRODUCTION

Most database query languages offer the expressive power of restricted first-order logic. Negative constraints in queries are typically expressed using negation as failure interpreted under the closed world assumption. Such negative constraints provide the only means of suppressing part of the database. For that purpose, however, negation as failure is quite limited. Suppression expressed by negation as failure falls down in cases where the negated atom does not naturally unify to other atoms in the query.

To capture more expressivity as far as the indicated need of suppression is concerned, we introduce in this paper a construct called *counterfactual exceptions*. A counterfactual exception is a negative hypothesis specified in the query. Our concern is to express counterfactual implication premises with a reading like "suppose it was not so that $\Pi \ldots$" and thereby suppress a part of the database corresponding to Π in the evaluation of the query. For instance, in the query *"I want to travel from A to B, but I refuse to fly"*, a counterfactual exception is embedded, "suppose it was not possible to fly", and thereby is expressed a need for a travel with any possible flights suppressed. Thus for a database DB of travel information and the query above, the idea is to evaluate *"I want to travel from A to B"* against DB updated with $\forall from, to : \neg flight(from, to)$. We are referring to a world without flights, but in any other respect as close as possible to the current world. The traditional treatment of this is by possible world counterfactual implication as originally suggested by Lewis in [10].

We propose a new mechanism treating negative hypotheses as exceptions to the predicates mentioned and achieve a much simpler semantics, declaratively as well as procedurally, while still preserving a mechanism which we believe to be useful for database applications. Definition and examples are given in section 2.

In some cases negation as failure may serve the same purpose as counterfactual exceptions and we have a close relation to possible world counterfactual implication. We compare other approaches and counterfactual exception in section 3. Querying can be described by an extension of the *modus ponens* rule and we show how this can be implemented in a vanilla-like interpreter — which in turn indicates that the mechanism can be implemented in more efficient ways without additional overhead. We cover inference and discuss an interpreter in sections 4 and 5. We show, in section 6, how this framework can be used to define a language with hypothetical implication goals with positive as well as negative hypotheses and show how the mechanism can be combined with negation as failure. — We have considered in this paper only the function free case, but our framework can be generalized to languages with function symbols. Finally we give a perspective and discuss related work in section 7.

[1] Roskilde University, Computer Science Dept., P.O.Box 260, DK-4000 Roskilde, Denmark

ECAI 96. *12th European Conference on Artificial Intelligence*
Edited by W. Wahlster
Published in 1996 by John Wiley & Sons, Ltd.

2 COUNTERFACTUAL EXCEPTIONS, DEFINITION AND EXAMPLES

We consider deductive databases (or Datalog programs) consisting of clauses of the form:

$$A_0 \leftarrow A_1 \wedge \ldots \wedge A_n$$

where $A_1, \ldots, A_n$ are function free atoms; variables are universally quantified within each clause. In the framework to be defined, we use the notation $\phi \twoheadrightarrow \psi$ to represent the informal query "if the exceptions expressed by ϕ were the case, can ψ hold?" The semantics is inherently difficult as the premise ϕ might be inconsistent with the given database; if we interpret $\phi \twoheadrightarrow \psi$ as classical implication it becomes uninteresting as anything follows from a false premise. Such counterfactual implications has been studied in detail by several authors [19, 10, 7, 16] and the most common way to achieve a reasonable interpretation is by the possible world semantics originally suggested by Lewis [10], saying that ψ holds in any world "maximally similar" with the real world (i.e., the database) in which ϕ holds. This construction involves also tracing the necessary "reasons" why ϕ could be the case and this is exactly where our mechanism differs; we illustrate the difference by means of examples in section 3 below.

We consider a restricted form of counterfactual implications, closed formulas of the form

$$\exists \cdots (\phi \twoheadrightarrow \psi)$$

with

$$\phi = (\forall \cdots \neg\phi_1) \wedge \cdots \wedge (\forall \cdots \neg\phi_n)$$

where $\phi_1, \ldots, \phi_n$ are atoms, ψ a conjunction of atoms; each subformula $\forall \cdots \neg\phi_i$ is called a *counterfactual exception*. Any variable quantified at the outermost level is said to be *global*, all other variables in the ϕ_i's are *local*. For simplicity we begin by considering the case without global variables and thus with ψ ground.

The semantics of the new implication arrow is defined by means of a generalization of the traditional fixpoint semantics for logical programs (see [11]). Given a database DB, we will recognize a formula $\phi \twoheadrightarrow \psi$ as true if and only if $\psi \in M^{\phi}_{DB}$ where M^{ϕ}_{DB} is *least model* for DB under the exceptions ϕ defined as follow,

$$M^{\phi}_{DB} = \text{lfp}(T^{\phi}_{DB})$$

that is, as the least fixed point of the function T^{ϕ}_{DB}, where T^{ϕ}_{DB} is the following generalized *consequence operator*.

$$T^{\phi}_{DB}(I) = \{\alpha \mid DB \text{ has a clause with a ground instance } \alpha \leftarrow \beta_1 \wedge \cdots \wedge \beta_k \text{ with } \beta_i \in I \text{ for all } i \text{ and } \phi \wedge \alpha \text{ is consistent}\}$$

In other words, we allow those immediate consequences of clauses in the database that do not conflict with the exceptions. Notice that consistency in the special case applied here is decidable; we will be more specific about this point below.

We illustrate the least model for the database DB_0 consisting the following three clauses,

$$\{p(X) \leftarrow q(X),\ p(a),\ q(b)\}.$$

Here we have that

$$M^{true}_{DB_0} = \{p(a), q(b), p(b)\}, \text{ while}$$
$$M^{\neg p(b)}_{DB_0} = \{p(a), q(b)\}, \text{ and}$$
$$M^{\forall Y \neg q(Y)}_{DB_0} = \{p(a)\}.$$

Notice in the first example that $true$ stands for an empty set of exceptions and in this case the least model coincides with the usual least model for the program DB.

The semantics for the general form of counterfactual implication can be defined by expressing the existential quantification of global variables at the meta-level as follows.

The formula $\exists X_1 \cdots X_n\, (\phi \twoheadrightarrow \psi)$ follows from a database DB whenever $\phi \twoheadrightarrow \psi$ has an instance $\phi' \twoheadrightarrow \psi'$ with $\psi' \in M^{\phi'}_{DB}$.

The use of negation as failure in queries with counterfactual exceptions can be justified by an equivalent semantic definition based on a generalized completion construction (cf. [1]). The clauses defining a given predicate are joined together as a bi-implicature formula and in our case we take into account the exceptions concerned with that predicate. We use the notation comp^{ϕ}_{DB} for the *completion* for DB under counterfactual exception ϕ. We leave out the definition and illustrate the construction by the following examples.

$$\text{comp}^{true}_{DB_0} = (p(Z_1) \leftrightarrow (Z_1 = X \wedge q(X)) \vee Z_1 = a) \wedge (q(Z_2) \leftrightarrow Z_2 = b)$$
$$\text{comp}^{\neg p(b)}_{DB_0} = (p(Z_1) \leftrightarrow ((Z_1 = X \wedge q(X)) \vee Z_1 = a) \wedge Z_1 \neq b) \wedge (q(Z_2) \leftrightarrow Z_2 = b)$$
$$\text{comp}^{\forall Y \neg q(Y)}_{DB_0} = (p(Z_1) \leftrightarrow (Z_1 = X \wedge q(X)) \vee Z_1 = a) \wedge (q(Z_2) \leftrightarrow (Z_2 = b \wedge \forall Y\ Z_2 \neq Y))$$

We can show the following equivalence between the completion and the model-based semantics.

$$M^{\phi}_{DB} = \{\alpha \mid \alpha \text{ is an atom such that } \text{comp}^{\phi}_{DB} \models \alpha\}$$

We define, thus, the following semantics for negation as failure.

not $(\phi \twoheadrightarrow \psi)$ holds iff $\psi \notin M^{\phi}_{DB}$

The following example illustrates the use of counterfactual exceptions in database applications.

Example 2.1 We will consider the following traveling information database DB.

$$\{travel(X,Y) \leftarrow link(X,Y),$$
$$travel(X,Y) \leftarrow link(X,Z) \wedge travel(Z,Y),$$
$$link(X,Y) \leftarrow train(X,Y),$$
$$link(X,Y) \leftarrow boat(X,Y),$$
$$link(X,Y) \leftarrow flight(X,Y),$$
$$flight(a,b), flight(b,c), flight(d,e), flight(e,a),$$
$$train(a,b), train(c,d), boat(b,c)\ \}$$

The counterfactual implication query "I want to travel from a to d, but I refuse to sail from b to c",

$$(\neg boat(b,c)) \twoheadrightarrow travel(a,d)$$

obviously succeeds since

$$\{flight(a,b), flight(b,c), train(c,d)\} \subseteq M^{\neg boat(b,c)}_{DB}.$$

The query "I want to travel from a to d, but I refuse to fly",

$$(\forall X, Y \neg flight(X,Y)) \twoheadrightarrow travel(a,d)$$

succeeds since

$$travel(a,d) \in M^{\forall X, Y \neg flight(X,Y)}_{DB} =$$
$$\{train(a,b), train(c,d), boat(b,c),$$
$$link(a,b), link(c,d), link(b,c),$$
$$travel(a,b), travel(c,d), travel(b,c),$$
$$travel(a,c), travel(a,d), travel(b,d)\ \}.$$

The following expresses "I want to travel from a to d, but I refuse to sail into the harbor of c".

$$(\forall X \neg boat(X,c)) \twoheadrightarrow travel(a,d)$$

We can show the use of global variables in the query "I want to travel from a to a place where I do not arrive by train".

$$\exists X((\forall Y \neg train(Y,X)) \twoheadrightarrow travel(a,X))$$

We should stress that counterfactual exceptions also may concern information which is not represented as facts in the database but implied from other facts. The following example may be relevant if you had all your luggage stolen in c on your last travel. "I want to travel from a to e, but I refuse to pass by c",

$$((\forall X \neg link(X,c)) \wedge (\forall X \neg link(c,X))) \twoheadrightarrow travel(a,e).$$

Having a careful look at the semantic definition, we observe that it is forbidden to apply any $link(\_,\_)$ via c in the evaluation of $travel(a,e)$ but it is still possible to use, say, $fligth(b,c)$ for other purposes than "linking" our traveler.

3 COUNTERFACTUAL EXCEPTIONS, COMPARED

As mentioned in the introduction, we may view counterfactual exceptions as a means of suppressing part of the database. In the following, we compare with negation as failure, which in some cases also may apply for this purpose, and with the different semantics provided by possible world counterfactual implication.

Exceptions provide a way to express that something is to be ignored, in the sense that it must not be *applied* in the evaluation of a goal. Negation as failure is something quite different because it states that something cannot be the case.

Consider the following database.

$$\{person(X) \leftarrow employee(X),\\ person(X) \leftarrow student(X), \ldots\}.$$

For this database, there is no difference between $(\forall X \neg student(X)) \twoheadrightarrow person(Y)$ and $person(X)$, not$student(X)$. However, this is a special case in the sense that the range of X is restricted by the unification in the query.

Continuing example 2.1, we notice that $(\forall X, Y \neg flight(X, Y)) \twoheadrightarrow travel(a, d)$ is independent of whether there exist flights (it succeeds even though there exists a travel from a to d which involves flights). On the other hand $travel(a, d)$, not$flight(X, Y)$ will of course fail. In section 6 we describe a language providing both negation as failure and exceptions, and we describe principles for an implementation.

For *possible world counterfactual implication* [10, 19, 7, 16], the main concern is consistency between the hypothesis and (a revision of) the database, thus $\neg q \twoheadrightarrow p$ is evaluated as follows:

- p is evaluated in possible worlds DB', obtained as $\neg q$ revisions of DB,
- thus, $\neg q$ is consistent with DB'

For *counterfactual exceptions*, we so to say narrow the scope to consistency with the proof, thus $\neg q \twoheadrightarrow p$ is:

- p is evaluated in DB such that only formulas consistent with $\neg q$ are applied,
- thus, p is evaluated in DB such that $\neg q$ is consistent with the proof.

We illustrate the difference between the two interpretations by means of the following database example.

$$DB = \{rains(copenhagen), cloudy(X) \leftarrow rains(X)\}$$

Within this database obviously $cloudy(copenhagen)$ holds. Now, consider the query Q_1, embedding the hypothesis that "it does not rain in Copenhagen":

$$Q_1\text{: } \neg rains(copenhagen) \twoheadrightarrow cloudy(copenhagen)$$

In the possible world interpretation the query expresses something like "Suppose it was not raining, would it then be cloudy in Copenhagen?" and the answer becomes "no", since the "world without rain in Copenhagen", (the hypothetical state of the database, that is) must be $DB' = \{cloudy(X) \leftarrow rains(X)\}$

By exception, the query Q_1 expresses "Even if it is not raining, can it possibly be cloudy in Copenhagen?". The answer again becomes "no", because it is not possible to derive $cloudy(copenhagen)$ without applying $rains(copenhagen)$.

Now, consider the query Q_2, expressing the hypothesis that "it is not cloudy in Copenhagen":

$$Q_2\text{: } \neg cloudy(copenhagen) \twoheadrightarrow rains(copenhagen)$$

By possible world interpretation the query expresses "Suppose it was not cloudy, would it then be raining in Copenhagen?". In this case we have two possible states with "no clouds over Copenhagen". The state $DB' = \{cloudy(X) \leftarrow rains(X)\}$ implies the answer "no" to Q_2, while the state $DB'' = \{rains(copenhagen)\}$ obviously leads to the answer "yes". Thus the answer to Q_2 by possible world interpretation would be "don't know".

By exception, on the contrary, the query expresses "Even if it is not cloudy, can it possibly be raining in Copenhagen?" and the answer is "yes" — we do not need to apply $cloudy(copenhagen)$ to derive $rains(copenhagen)$.

4 INFERENCE UNDER COUNTERFACTUAL EXCEPTIONS

As noticed above, a query $\phi \twoheadrightarrow \psi$ with exception ϕ means that only formulas that are consistent with ϕ should be used in the proof of ψ. We can use this principle to formalize a deductive system for such queries by the following modification of *modus ponens*.

Assume a fixed exception ϕ (posed in the query) and a database DB. For simplicity, we give the rule in a form suited for the ground case; nonground clauses and queries are covered by considering the sets of their ground instances.

$$\frac{\beta \qquad \beta \rightarrow \alpha}{\alpha} \quad \text{when } \phi \wedge \alpha \text{ consistent}$$

The resulting deductive system is obviously sound and complete, i.e., $\{\psi \mid DB \vdash^{\phi} \psi\} = M^{\phi}_{DB}$ where $\vdash^{\phi}$ is the proof relation defined by this deductive system.

5 AN INTERPRETER CAPTURING EXCEPTIONS

The deductive system can be made into a running interpreter by a modification of the well-known Vanilla interpreter. The underlying Prolog semantics provides a correct handling of variables in queries and clauses, which we had abstracted away in the deductive system.

The clauses of the database are represented by a predicate `clause(`*head*`, `*body*`)` with variables given as Prolog variables. The exceptions appear as an explicit argument in the interpreter in order to provide the necessary communication through the global variables.

```
% prove( φ, ψ ) if and only if φ ↠ ψ
prove(_, true):- !.
prove(Cf, (A,B)):-
    !, prove(Cf, A), prove(Cf, B).
prove(Cf, A) :-
    clause(A,B),
    consistent(A, Cf),
    prove(Cf, B).
```

The completion semantics gives a hint for an implementation of the consistency condition. The selected atom `A` must satisfy a condition of non-unifiability with each atom appearing negatively in the exceptions. To this end, we use a declarative `dif(-,-)` predicate as is found in, e.g., Sicstus Prolog [18]. A call `dif(`s`,`t`)` will delay a test for syntactic inequality until the moment that s and t are sufficiently instantiated to tell them either identical or non-unifiable; this provides a "lazy" evaluation strategy whose overall behaviour is consistent with the ground-case deductive system above.

Each exception gives rise to a condition derived in the following way; a straightforward call to `dif` between two atoms is not sufficient. We have to distinguish between local and global variables and also take into account any possible aliasing expressed by local variables. We analyze the arguments in each exception's atom in the following way.

- An argument which is a constant `c` or global variable `G` must always be different from the corresponding argument in the selected atom. This amounts to a test `dif(c,X)` or `dif(G,X)` where `X` refers to the corresponding argument in the selected atom `A`.
- A local variable occurring only once will always unify with the corresponding argument of the selected atom. This corresponds to always `fail`.
- A local variable which occurs as the ith as well as the jth argument implies a test `dif(`X_i`,`X_j`)` where X_i and X_j refers to the corresponding arguments of the selected atom.

Finally, these tests are merged together in a single call to `dif(-,-)` to express their disjunction.

Consider, as an example, an exception $\forall L_1 L_2 \neg p(a, G, L_1, L_2, L_2)$ where a is constant and G a global variable. When `A` refers to the selected atom, the consistency test can be implemented by the following piece of Prolog code.

```
A = p(X1, X2, X3, X4, X5)
  -> dif((X1,X2,X4), (a,G,X5)) ; true
```

The syntax $\cdots$ `->` $\cdots$ `,` $\cdots$ denotes an if-then-else construction in Prolog.

In case of an "exhaustive" exception such as $\forall X, Y \neg flight(X, Y)$, the consistency test falls down to the following.

```
A = flight(_,_) -> fail ; true
```

The answers provided by this interpreter consists of bindings to variables together with a (perhaps empty) collection of unresolved `dif(-,-)` calls. Under the assumption of infinitely many constant symbols, this represents a satisfiable set of equations and inequalities, so the interpreter is sound. Concerned with completeness, it is easy to prove to following statement: "If it were the case that Prolog had used breadth-first search for selecting clauses, then this meta-interpreter would have been logically complete."

The metainterpreter is suited for prototyping and it is easy to imagine a much more efficient implementation as a straightforward modification of a Datalog interpreter written in some low-level language. Negation as failure will work immediately for queries without global variables; we consider the case with global variables in section 6 below.

Example 5.1 Efficient Prolog programs can be produced by partially evaluating the meta-interpreter with respect to a particular set of exceptions. For each clause in the original database, we get a new clause extended with code for those exceptions that concern the predicate in the head of the clause.

We consider first a case without global variables occurring in the exception. We continue example 2.1 and consider queries of the following form.

$$(\forall L \neg link(L, c)) \twoheadrightarrow \cdots$$

Each clause concerned with the $link$ predicate will be equipped with a `dif` test as shown in the following example.

```
link(X,Y):- dif(Y,c), boat(X,Y).
```

Global variables are treated similarly to constants, except that an additional argument must be added to each predicate to provide communication in global variables. Considering queries of the following form,

$$\exists G((\forall L \neg link(L, G)) \twoheadrightarrow \cdots G \cdots,$$

the partially evaluated program will now include the following clause.

```
link(X,Y,G):- dif(Y,G), boat(X,Y).
```

6 NEGATIVE COUNTERFACTUALS IN HYPOTHETICAL IMPLICATION GOALS

We will show here how our notion of counterfactual exceptions can be integrated in a language with hypothetical implication goals in the body of a clauses.

An operational semantics for implication goals in a language without negative constructs can be derived from the classical deduction theorem (see, e.g., [3]),

$$\Gamma \models \phi \rightarrow \psi \quad \text{iff} \quad \Gamma \cup \{\phi\} \models \psi.$$

Thus, in order to prove $\phi \rightarrow \psi$, extend the program with the clauses contained in ϕ and prove ψ in the usual way. This principles has been used in the design of several programming languages, we can refer to [4, 13, 5, 6, 14, 12].

In our language, we can have exceptions as well as positive clauses as hypotheses and we have also included negation as failure. The overall structure of the language is given by the following grammar.

$$\begin{array}{l}\langle\text{database}\rangle ::= \quad \langle\text{clause}\rangle \cdots \langle\text{clause}\rangle \ \langle\text{exception}\rangle \cdots \langle\text{exception}\rangle \\ \langle\text{clause}\rangle ::= \quad \langle\text{atom}\rangle \leftarrow \langle\text{goal}\rangle \cdots \langle\text{goal}\rangle \\ \langle\text{goal}\rangle ::= \quad \langle\text{atom}\rangle \mid \langle\text{database}\rangle \twoheadrightarrow \langle\text{goal}\rangle \cdots \langle\text{goal}\rangle \mid \text{not } \langle\text{goal}\rangle\end{array}$$

Global variables in an exception may be quantified at the level of the surrounding implication goal or at the clause level.

The database component of a hypothetical implication goal will serve as an extension to the current database in such a way that the order in which the clauses and exceptions have been added is preserved. A given set of hypotheses is written as $D_i C_i$ where D_i refers to clauses, C_i to the exceptions. We can thus write the database available at the nth level of embedded implication as $D_0C_0 \bullet \cdots \bullet D_nC_n$ where $\bullet$ is a sequence constructor, D_0C_0 is the initial database.

The semantics is defined by a deduction system that determines a relation $\vdash$ between such layered databases and goals; as before, the deduction system is described for ground queries and clauses, the general case is covered by considered the set of all their ground instances.

The rule for hypothetical implication goals shifts the premises into the database component.

$$\frac{D_0C_0 \bullet \cdots \bullet D_nC_n \bullet D_{n+1}C_{n+1} \vdash \psi}{D_0C_0 \bullet \cdots \bullet D_nC_n \vdash D_{n+1}C_{n+1} \twoheadrightarrow \psi}$$

In the modified *modus ponens* rule we let a given clause be restricted by exceptions that are more recent that the clause but not by previous ones that the clause is supposed to override.

$$\frac{D_0C_0 \bullet \cdots \bullet D_nC_n \vdash \beta \qquad D_iC_i \vdash \beta \rightarrow \alpha}{D_0C_0 \bullet \cdots \bullet D_nC_n \vdash \alpha}$$

when $C_i \wedge \cdots \wedge C_n \wedge \alpha$ consistent, $1 \leq i \leq n$

Negation as failure is covered as follows.

$$DB \vdash \text{not } \phi \quad \text{iff} \quad \text{we do not have} \quad DB \vdash \phi$$

We can construct an implementation for this language by a straightforward extension of the Vanilla-like interpreter shown in section 5. The current database should be passed as an additional argument to the `prove` predicate. A ground representation (i.e., with constants to name variables) is recommended here whereas variables in the query should appear as Prolog variables; the *modus ponens rule* must, then, include an explicit instance-of condition to create variants of database clauses with new Prolog variables. Notice that global variables may appear as Prolog variables in the database argument. For a background on these rather technical matters, see [8].

In the implementation of negation-as-failure, we use co-routines in order to avoid the unsound behaviour demonstrated by most Prolog systems. The following Prolog clause, which uses special Sicstus Prolog constructs, will do the job.

```
prove((not Goal), DB):-
 when(ground((Goal,DB)), \+ prove(Goal,DB)).
```

An invocation of it will delay the call in the body until it becomes ground; "`\+`" is Sicstus Prolog's notation for the traditional approximation of negation as failure which is sound for ground goals. If end of execution is reached and there are still such delayed calls, the computation is characterized as floundering and no answer is provided. However, if all clauses are range restricted this cannot happen.

A concrete syntax and an actual implementation of the language has been developed and described in [15].

7 PERSPECTIVES AND RELATED WORK

We have suggested the use of counterfactual exceptions as hypotheses in database queries as a means to limit the relevant answers by suppressing certain parts of the database. We have also shown how this mechanism fits into a language with hypothetical implication goals and negation as failure. The approach is related to possible world counterfactual implication [10], but is characterized by a straightforward semantics and implementation.

Logic programming with exceptions have been studied previously by [9]. They consider only exceptions being part the program, not as hypotheses likely to be inconsistent the actual database. On the other hand, they can define exceptions by arbitrary clauses. This implies that answer sets are not unique, as is the case in our approach. The clauses we obtain by partially evaluating our interpreter, example 5.1, appear as optimized versions of the outcome of the transformation suggested by [9], improved also in the sense, that the use of the `dif(-,-)` technology makes our programs terminate in meaningful states in cases where the corresponding programs of [9] give up as floundering.

Metalevel negation as featured in Reflective Prolog [2] can be used to express our counterfactual exceptions as well as the mechanism of [9]. The programmer supplies, at the metalevel, "solve" rules which extends the basic interpretation strategy for the object language together with "solve_not" rules. A goal is consider to hold if can be proved by the "solve" predicate unless "solve_not" also holds for it. In the present context, we can consider Reflective Prolog a powerful implementation language, which leaves it up to the programmer to decide upon the logical semantics of the object language.

We notice that the referenced approaches [9, 2] can handle a substantial portion of Reiter's default logic [17], where we cover only a small part. We have concentrated on a conceptually and technically simpler mechanism while still retaining an interesting expressibility with respect to database querying. The use of global variables appearing simultaneously in the exceptions as well as in the positive part of the query seems to be unique to our approach.

In section 6 we mentioned a number of other languages with hypothetical implication goals but we are not aware of other approaches with exceptions in the hypotheses. It seems possible that our approach can be combined with that of [9] to include the more general kinds of exceptions as hypothesis.

REFERENCES

[1] Clark, K.L., Negation as failure. *Logic and Data Bases*, Gallaire, H., and Minker, J. (eds.), Plenum Press, pp. 293–322, 1978.

[2] Costantini, S., and Lanzarone, G.A., Metalevel negation and nonmonotonic reasoning. *Methods of Logic in Computer Science* 1, pp. 111–140, 1994.

[3] Enderton, H.B., *A Mathematical Introduction to Logic*. Academic Press, 1972.

[4] Gabbay, D.M. and Reyle, U., N-Prolog: An extension of Prolog with hypothetical implications. *Journal of Logic Programming* 2, pp. 319–355, 1984.

[5] Giordano, L. and Martelli, A., A modal reconstruction of blocks and modules in logic programming. *International Logic Programming Symposium*, 1991.

[6] Giordano, L., Martelli, A., and Rossi, G., Extending Horn clause logic with implication goals. *Theoretical Computer Science*, 1991.

[7] Ginsberg, L.M., Counterfactuals. *Artificial Intelligence* 30, pp. 35–79, 1986.

[8] Hill, P.M. and Gallagher, J.P., Meta-programming in Logic Programming. To be published in Volume V of *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press. Currently available as *Research Report Series* 94.22, University of Leeds, School of Computer Studies, 1994.

[9] Kowalski, R.A., and Sadri, F., Logic programming with exceptions. *Proc. of Eighth International Conference on Logic Programming*, MIT Press, pp. 598–613, 1991.

[10] Lewis, D, *Counterfactuals*. Harward University Press, 1973.

[11] Lloyd, J.W., *Foundations of logic programming*, Second, extended edition. Springer-Verlag, 1987.

[12] Miller, D., Lexical scoping as universal quantification, *Proc. of Sixth International Conference on Logic Programming*, MIT Press, pp. 268–283, 1989.

[13] Monteiro, L. and Porto, A., Contextual Logic Programming, *Proc. of Sixth International Conference on Logic Programming*, MIT Press, pp. 284–302, 1989.

[14] Nait Abdallah, M.A., Ions and local definitions in logic programming, *Lecture Notes in Computer Science* 210, pp. 60–72, Springer-Verlag, 1986.

[15] Ochotorena, C., *A database language with hypothetical implementation goals*. Unpublished report. Roskilde University, 1995.

[16] Pereira, L.M., Aparício, J.N., and Alfares, J.J., Counterfactual reasoning based on revising assumptions. Logic Programming, Proceedings of the 1991 Internal Symposium, MIT Press 1991.

[17] Reiter, R., A logic for default reasoning. *Artificial Intelligence* 13, pp. 81–132, 1980.

[18] *SICStus Prolog user's manual*. Version 3 #0, SICS, Swedish Institute of Computer Science, 1995.

[19] Stalnaker, R., A theory of conditionals. *Studies in Logical Theory*, ed. Reicher, N., Oxford University Press, 1968.