EOLC: Efficiently Modelling Inconsistency for Commonsense Reasoning

Rajesh Kumar[1], Ashish Tiwari[2], and Bruce H. Krogh[1]

[1] Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
rajeshk@ece.cmu.edu, krogh@ece.cmu.edu
[2] SRI International
Menlo Park, CA 94025-3493, USA
tiwari@csl.sri.com

Abstract. This paper presents *EOLC*, a declarative rule language for commonsense reasoning incorporating non-monotonicity using a four-valued logic, to explicitly model overspecified information, priorities among rules using an `overrides` predicate, arithmetic constraints, and optimization through an ordering operator `rank`. *EOLC* also supports the recursive definition of rules. We give the declarative semantics of *EOLC* and present results about the models of *EOLC* programs and the complexity of inferencing in *EOLC*.

1 Introduction

In 1959, McCarthy said a program would exhibit "commonsense" if it "automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told and what it already knows" [1]. The formalism used for knowledge representation determines the extent to which a class of programs can perform such reasoning. Efforts towards commonsense reasoning began with work on different forms of negation [2, 3], circumscription [4], default logic [5], prioritized logic programs [6] and defeasible logic [7], one of the most recent being courteous logic programs [8]. Priorities handle conflicts between rules and make it possible to use naturally available prioritization information. This paper presents *EOLC*, a rule language with a rich set of knowledge representation primitives.[3] *EOLC* extends features of previous languages formalisms, with four-valued logic, priorities, and rank-ordered rules to support commonsense reasoning in a large number of applications.

EOLC works in a four-valued logic explicitly modeling inconsistency in knowledge. Inconsistencies arise naturally in an evolving knowledge base. Knowledge representation formalisms that do not work in four-valued logic cannot distinguish between: (i) neither a nor $\neg a$ have supporting evidence – a is unknown; and (ii) both a and $\neg a$ have supporting evidence – a is overspecified. General

[3] *EOLC* stands for *Epistemic Ontology Language with Constraints*, since *EOLC* is meant to specify the reasoning or epistemic part of a knowledge base.

and extended logic programs work in two-valued logic [9] while formalisms like courteous logic programs and defeasible logic combine the cases (i) and (ii) [8, 7]. Paraconsistent logics distinguish between (i) and (ii) [10], but *EOLC* supports this distinction in the presence of rule priorities, arithmetic constraints and a form of aggregation constraints. Section 3 illustrates why explicitly modelling such overspecification is necessary for commonsense reasoning.

Courteous logic programs [8], which support explicit and implicit negation and priorities among rules, control complexity by working in a domain without function symbols or circular dependencies among atoms – the acyclic, Datalog-restricted domain. In many scenarios, recursion, however, arises naturally. General logic programs and variants such as extended logic programs support recursive definitions [9], but no priorities, while prioritized and defeasible logics have severe complexities [11]. The language proposed in this paper, *EOLC*, removes the acyclicity restriction and efficiently supports recursion in the presence of explicit negation, priorities, and constraints.

Many applications require support for arithmetic constraints [12]. *EOLC* supports arithmetic constraints under some restrictions. An *aggregation operator*, `rank`, in *EOLC* enables the selection of a particular (optimal) solution from among a set of possible solutions based on arithmetic expressions. Aggregation operators has been studied in the work on constraint logic programs with optimization [13, 14] and preference logic programming [15]. *EOLC* goes beyond the capabilities of these formalisms by supporting multiple levels of selection through the use of the `rank` operator in different rules.

EOLC has a simple declarative semantics, to enable non-experts to comfortably specify rule sets, and an efficient inferencing algorithm. By removing the acyclicity assumption, using a four-valued logic, and supporting constraints with aggregation to select desirable solutions, *EOLC* is ideally suited to support commonsense reasoning in many application domains, including verification management [16], which motivated *EOLC*.

This paper is organized as follows. Section 2 gives the syntax of *EOLC* and the semantics of *EOLC* programs in terms of `models`. Section 4 shows that each *EOLC* program has a unique maximal model under the restrictions on recursion and constraints that we assume, describes the algorithm to compute `models` and presents the complexity of inferencing in *EOLC*. We conclude the paper with a summary of the contributions of *EOLC* and directions for future work.

2 The *EOLC* Language

Mutually disjoint sets of symbols specify the *object constants* ($\mathcal{O}$), *variables* ($\mathcal{V}$) that range over object constants, and *predicates* ($\mathcal{P}$) in an *EOLC* program. We adopt the Prolog convention in that variables start with a capital letter and constants are in small case. An *assignment* maps variables to object constants.

An *atom* has the form $P^n(t_1, t_2, \ldots t_n)$ where P^n is an n-ary predicate and $t_1, t_2, \ldots, t_n$ are variables or constants. Atoms in which there are no occurrences of variables are called *ground atoms.* A *literal* is of the form p or $\neg p$, where p

is an atom. A literal of the form p ($\neg p$) is called a positive (negative) literal. Ground literals evaluate to one of four truth values *true, false*, *unknown* ($\bot$), *overspecified* ($\top$), determined by the semantics defined in section 4.

For *EOLC*, the constraint domain includes functions $*, +, -$, and predicates $\mathcal{P}_{EOLC} = \{=, <, \leq, >, \geq\}$ with their usual arithmetic meanings. Constraints are formulas with predicates $\mathcal{P}_{EOLC}$ and without quantifiers. Formulas in which only $*, +, -$ occur and $\mathcal{P}_{EOLC}$ do not occur are called expressions. Satisfiability in this constraint domain of arithmetic over integers is undecidable [17]. The restrictions we put on variables in constraints and expressions ensures that only ground constraint atoms in this constraint domain need to be evaluated. Ground constraints evaluate to *true* or *false* (and never to $\bot$ or $\top$). Ground atoms of only un-interpreted predicates, $\mathcal{P} \backslash \mathcal{P}_{EOLC}$, may evaluate to $\top, \bot$.

Labelled rule: A *labelled rule* (R) is of the form:

$$\langle lab \rangle L_0 : L_1, L_2, \ldots L_n ; \texttt{not} L_{n+1}, \ldots, \texttt{not} L_m ;$$
$$c_1, c_2, \ldots, c_k ; \texttt{rank}(c).$$

where each $L_i, i \in \{1, \ldots, m\}$ is a literal, c_i are constraints and 'c' within the scope of `rank` is an expression. When the variables in c are instantiated, c evaluates to a numeric value. A *ground instance* of a labelled rule is one in which all variables have been assigned to object constants. Ground instances of rule R are ordered or ranked according to the value of the expression c and this ordering is used in defining the semantics of the *EOLC* program. $\langle lab \rangle$ is an optional string label. All rules without labels are treated as having a default label "empty_label".

A rule that has no constraints on the right hand side is called constraint-free. $labels(l, R)$ will denote that 'l' is the label of a rule R. For a rule R, we shall refer to $\{L_1, \ldots, L_n\}$ as *pos_body*(R) (the positive part of the body of R), and $\{L_{n+1}, \ldots, L_m\}$ as *neg_body*(R) (the negative part of the body of R). For a rule R, *head*(R) will denote the literal L_0 and *body*(R) will denote the literal(s) $\{L_1 \ldots L_m\}$.

Priority Ordering: A *priority ordering* among rules is a set of declarations of the form `overrides`(lab_1, lab_2) where lab_i are labels of some rules. `overrides` must satisfy a strict partial ordering relation, i.e., `overrides` is irreflexive and transitive.

EOLC program: An *EOLC* program E consists of a set of labelled rules and a priority ordering between the rules. The set of all ground atoms constructed using predicates and constants in E, is called its *Herbrand base*, $\mathcal{H}_E$. For *EOLC* program E, the instantiated *EOLC* program, E^{instd}, is the set of all the instantiations of rules in E along with the prioritization predicates of E. For an *EOLC* program E, E^{instd} is bounded (details in [18], Section 7).

Interpretation: A tuple $\langle S, X \rangle$, where S and X are sets of literals, gives a 4-valued interpretation to all ground literals in an *EOLC* program. S defines the true and false literals, literals in X are interpreted as $\top$ and the remaining literals are interpreted as $\bot$.

Enabled ground rule instance: Consider a ground rule $\hat{R}$

$\langle lab \rangle L_0 : L_1, L_2, \ldots L_n; \texttt{not} L_{n+1}, \ldots, \texttt{not} L_m;$
$c_1, c_2, \ldots, c_k; \texttt{rank}(c).$

Let $pos_R = \{a | a, \neg a \in pos\_body(\hat{R})\}$. $\hat{R}$ is *enabled* in an interpretation $\langle S, X \rangle$, iff, (pos_body($\hat{R}$) $- pos_R) \subseteq S$, neg_body($\hat{R}$) $\cap (S \cup X) = \varnothing$, $c_1 \ldots c_k$ evaluate to *true*, and $pos_R \subseteq X$.

Intuitively, a rule is enabled when every literal in its body evaluates to *true* or $\top$. Here `not` has the semantics of "negation by failure" whereby $\texttt{not}(\top) = \bot$ (hence $\forall b \in neg\_body(\hat{R}), b \notin X$) and $\texttt{not}(true) = false$.

Well-grounded rule instances: A rule instance $\hat{R}$ in *EOLC* program E, is well-grounded if, (i) body($\hat{R}$) is empty, i.e., $\hat{R}$ is an asserted atom, OR (ii) each literal in $pos\_body(\hat{R})$ occurs as the head of some well-grounded rule instance $\hat{R}'$.

From now on we will use *ground instance* of a rule to mean *well-grounded instance* of a rule, i.e., we're only interested in well-grounded instances.

Ordering: Let $\mathcal{G}_R$ be the set of all ground instances of rule R that are enabled in an interpretation $\langle S, X \rangle$. An ordering is defined on the elements of $\mathcal{G}_R$ as: for $R_1, R_2 \in \mathcal{G}_R$, if $c_{R_1} \geq c_{R_2}$ then $R_1 \geq R_2$ in the ordering, where c_{R_1}, c_{R_2} are the values of the expression c within the scope of *rank* in R_1, R_2.
$\hat{R} \in \mathcal{G}_R$ lies at the top of its ordering, denoted by $\hat{R} \in \mathcal{R}^T$ iff (i) for all rule instances $\hat{R}' \in \mathcal{G}_R$, $\hat{R} \geq \hat{R}'$, OR, (ii) R does not have a rank-constraint in which case $\forall \hat{R} \in \mathcal{G}_R$, $\hat{R} \in \mathcal{R}^T$. Since $\mathcal{R}^T \subseteq \mathcal{G}_R$, each rule instance in $\mathcal{R}^T$ is enabled in $\langle S, X \rangle$.

We only need to compare ground instance of the same rule with each other and each rule R has a set $\mathcal{R}^T$ in an interpretation $\langle S, X \rangle$, as defined above. The following restrictions are imposed on labelled rules.

1. Variables in constraints c_i and expression c must occur in L_1 through L_m as terms.[4]
2. For any rule R, s.t., *head*(R) corresponds to a predicate with integer valued arguments, the body of R cannot be empty. Further, variables corresponding to integer valued attributes of *head*(R) must occur in the body of R.

The above restrictions ensure that when the variables in $L_0 \ldots L_m$ in R are assigned, all variables in the constraint expressions in R also get assigned and the constraints can be evaluated. Under the restrictions one cannot define unary predicates over integers in *EOLC*; for example, the rule "even(x): even(y), y = x-2." violates condition 1. Since the number of constants that occur in the *EOLC* program is finite, the above restrictions ensure that the number of possible satisfiable instantiations of each rule is finite. Hence, for a rule R, $\mathcal{R}^T \neq \varnothing$ if $\mathcal{G}_R \neq \varnothing$. Thus, constraints in *EOLC* support arithmetic statements on numeric attributes of entities. [5]

[4] These terms occur in literals corresponding to predicates that are integer valued attributes of entities in the asserted part of the knowledge base

[5] This is not a major restriction since we may have a separate "constraints library" with definitions of predicates such as "even". We can then use predicates defined in the library is rule bodies in an *EOLC* program.

most_reliable(Mdl, Sys) : implements(Mdl, Sys), reliability(Mdl, X); `rank`(X).
$\langle l_1 \rangle$ satisfies(M, C): satisfies_fact(M, C).
$\langle l_1 \rangle$ satisfies(M, C): submodel(M', M), satisfies(M', C).
submodel(M, M'): submodel_fact(M, M').
submodel(M, M'): submodel_fact(M, M''), submodel(M'', M'), $M \neq M''$.
satisfies_fact(m_2, c_2).
submodel_fact(m_1, m_3).
$\langle l_2 \rangle\neg$satisfies(M, C): failtest(M, C).
failtest(m_1, c_1).
implements(m_1, s_1).
`overrides`(l_2, l_1).

Fig. 1. Example *EOLC* program.

The above definitions are clarified with the *EOLC* program E_1 (example 1). E_1 states that a model satisfies a constraint if it is known to do so or if a submodel of it satisfies that constraint. No model can be a submodel of itself, a model is a submodel of another if the fact is known and the submodel relation is transitive. Rules labelled with l_2 have higher priority than rules labelled with l_1. The predicates `submodel` and `satisfies` have been recursively defined. The definition of `most_reliable(Mdl, Sys)` using `rank` states that the model of a system with the highest reliability index is the most reliable model.

An *atom-dependency graph* of an *EOLC* program E has all ground atoms in $\mathcal{H}_E$ as nodes and for two nodes A and B, there is a directed edge from A to B if there is a ground rule with head as A or ¬A, and the body contains B or ¬B ([9], page 8). A topological sort of a directed graph is a sequence of nodes $n_1 \ldots n_m$, s.t., $\nexists$ $edge(n_i, n_j), i > j$. Our inferencing algorithm works bottom up, considering literals according to a *stratification* order, which intuitively means that a literal is considered only after its dependencies have already been considered.

Definition 1. *(Stratification of atoms) A sequence of all the ground atoms in E, $\rho = p_1, p_2, \ldots p_n$, is a stratification of the atoms in E if ρ is a reverse topological sort of the atom dependency graph of E.*

A stratification exists if the atom-dependency graph is acyclic. When there are recursively defined predicates, we will use a stratification of a modified atom-dependency graph, where the cliques involving the recursive predicates are replaced by supernodes. The notation $p < q$ ($p > q$) *in* ρ will denote that p is before (after) q in stratification ρ. For a literal p, we will use the term *level of* p to mean the maximum length of a path from the atom in p to a node without children in the modified atom-dependency graph of the *EOLC* program. A set of literals, S, is consistent if there is no atom p, s.t., both $p, \neg p \in S$. The semantics of *EOLC* are defined in terms of `models` of *EOLC* programs.

Definition 2. *(Model of EOLC program E) An interpretation,* $\langle S, X\rangle$*, is a model of* E*, if:*

$\forall p \in S$*: (condition (a))*

(i) $\exists$ *ground instance* $\hat{R}$ *of some rule* R*, s.t.,* $head(\hat{R}) = p$*, and* $\hat{R} \in \mathcal{R}^T$

(ii) $\forall$ *instances,* $\hat{R}'$ *of any rule* R'*, s.t.,* $head(\hat{R}') = \neg p$ *and* $\hat{R}' \in \mathcal{R}'^T$*,* $\exists$ *instance* $\hat{R}''$ *of a rule* R''*, s.t.,* $\hat{R}'' \in \mathcal{R}''^T$ *and* $head(\hat{R}'') = p$*, s.t.,* $\hat{R}''$ *overrides* $\hat{R}'$

$\forall x \in X$*: (condition (b))*

(i) $\exists \hat{R}_1 \in \mathcal{R}_1^T$*,* $head(\hat{R}_1) = x$*,* $\nexists$ *rule instance* $\hat{R}_2 \in \mathcal{R}_2^T$ *with* $head(\hat{R}_2) = \neg x$*, s.t.,* $\hat{R}_2$ *overrides* $\hat{R}_1$

(ii) $\exists \hat{R}_1 \in \mathcal{R}_1^T$*,* $head(\hat{R}_1) = \neg x$*,* $\nexists$ *rule instance* $\hat{R}_2 \in \mathcal{R}_2^T$ *with* $head(\hat{R}_2) = x$*, s.t,* $\hat{R}_2$ *overrides* $\hat{R}_1$

Recall that by definition of $\mathcal{R}^T$ for any rule R, all $\hat{R} \in \mathcal{R}^T$ are enabled in $\langle S, X\rangle$. Intuitively, whether a literal p is true depends on whether the rules with head p that are enabled are able to defeat the rules with head $\neg p$. This is in the spirit of argumentative semantics for non-monotonic reasoning [19].

Definition 3. *(Maximal model)* $\langle S, X\rangle$ *is a* maximal *model of an EOLC program E iff,* $\forall p$ *in Herbrand base of E (i) if condition (a) holds on p in* $\langle S, X\rangle$ *then* $p \in S$*, and (ii) if condition (b) holds on p in* $\langle S, X\rangle$ *then* $p \in X$*.*

Well-grounded rule instances are used to define a model is so that literals in S are derived from ground facts in the program and the model is *supported* is the sense of Apt et al. [20]. The *EOLC* program consisting of the rules "$A(a) : c, A(b)$.", "$A(b) : c, A(a)$." and "c." has the maximal model: $M_1 = \langle\{c\}, \varnothing\rangle$. $M_2 = \langle\{A(a), A(b), c\}, \varnothing\rangle$ is not a model since the two rule instances with $A(a), A(b)$ are not well-grounded. For program E, all well-grounded rule instances in E^{instd} may be computed in $O(l_{max})$ passes over E^{instd} using the definition of well-grounded rule instances. The definition of a maximal model above captures the notion of stability (Gelfond [21]).

Lemma 1. *If* $\langle S, X\rangle$ *is a model of an EOLC program E then S is consistent.*

Proof: Follows from the definition of a model □

The condition on X, where literals in X evaluate to $\top$, intuitively states that there was conflicting information and the prioritization was insufficient to resolve the conflict.

3 *EOLC* as a Knowledge Representation Language

We illustrate the usefulness of *EOLC* for representing knowledge in commonsense reasoning applications. In *EOLC* explicit negative information is represented using ¬. For example, in a verification scenario ¬ *satisfies(system model, constraint)* is quite different from `not` *satisfies(system model, constraint)*; the former implies that the property has been refuted while the latter implies that not enough information is present to decide whether the model satisfies the property.

```
allow(X, fileServer): system_administrator(X).
                ⋮
¬ allow(X, fileServer): outside_carnegie_mellon(X).
                ⋮
ask_for_maintenance_password(X, fileServer):
                allow(X, fileServer), ¬ allow(X, fileServer) .
                ⋮
permit_maintenance(X, fileServer):
                maintenance_password_verifies(X, fileServer).
```

Fig. 2. Access control.

```
                ⋮
resolve(Mdl, C): satisfies(Mdl, C), ¬ satisfies(Mdl, C).
                ⋮
```

Fig. 3. Overspecification in a verification scenario.

EOLC works in a four-valued logic and makes the distinction between: (i) neither a nor $\neg a$ have supporting evidence – a is unknown; and (ii) both a, $\neg a$ have supporting evidence – a is overspecified. To see how this is useful consider the example in figure 3.

Figure 3 shows a part of an *EOLC* program for controlling access to a file server. A user that is a system administrator should be allowed access but a user trying to login from outside the campus should be denied access, since this is a highly secured machine. These rules may be added incrementally, maybe by different people, as the knowledge base evolves. Now there is an inconsistency in the rule specification since a system administrator trying to login from outside the campus network will not have access. Since *allow(X, fileServer)* would be overspecified, the system administrator is asked for the maintenance password to gain access.

Similarly, in a verification scenario, as in figure 1, one could have a rule (see figure 3) saying that if *satisfies(Mdl, C)* is overspecified then a verification engineer needs to step in and resolve the issue. A literal may become overspecified in a continuously evolving knowledge base and one needs the logic value $\top$ in the logic to model such situations.

The `rank` construct in *EOLC* may be used to pick out a desirable solution from among a set of solutions that fulfill a criteria. Figure 1 shows how a component model with the greatest reliability index may be picked from among the models that implement a particular component of a system architecture. *EOLC* goes beyond current capabilities [13, 22, 14, 15] by supporting multiple levels of

```
                ⋮
select(Mdl, Sys) : most_reliable(Mdl, Sys), cost(Mdl, C);
                                              rank(−C).
                ⋮
most_reliable(Mdl, Sys) :
            implements(Mdl, Sys), reliability(Mdl, X);
                                              rank(X).
```

Fig. 4. Multiple levels of selection.

selection using `rank` in different rules. Figure 3 shows how the least cost model may be selected from among those that have the greatest reliability index.

EOLC also supports a partially ordered prioritization among rules (Figure 1 shows an illustration). The need for priorities is well-established in the literature [5–8]. Section 5 presents a more rigorous comparison of *EOLC* with other recent non-monotonic formalisms.

4 Inferencing in *EOLC*

We consider the problem of deciding if a query, which is a ground literal p, is *true* in a given *EOLC* program. We impose the following restrictions
R1: Recursively defined predicates are partially ordered – predicate A comes before B if a definition of B uses A. We disallow mutual recursion, such as

A(X, Y) : B(X, Z), A(Z, Y).

B(X, Y) : A(X, Z), B(Z, Y).

R2: We disallow recursive definitions of the form

A(X, Y) : ... ; ... `not` A(Z, Y).

A(X, Y) : ... , ¬ A(Z, Y) ... .

R3: The definition of recursive predicates do not involve aggregation constraints using the `rank` operator since such a definition is counterintuitive.

R1 implies that cycles in the atom dependency graph involve ground instances of the same predicate and ground instances of recursively defined predicates form cliques. If each clique is collapsed to a 'super-node' then the resulting graph is acyclic

R2 implies that literals involving recursively defined predicates can be monotonically added to the model, i.e., the truth (falsity) of one instance of a predicate P, will not cause an earlier deduced instance to become *false* (*true*)

In *EOLC* program E_1 (figure 1), the predicates `satisfies` and `submodel` are recursively defined but there is no circular recursion. Figure 5 shows a part of the atom-dependency graph for program E_1. The graph has been drawn with connectors (circles labelled with letters A, B, ...) for the sake of clarity. For example, the connector 'B' denotes that there is an edge from 'submodel(m_1,

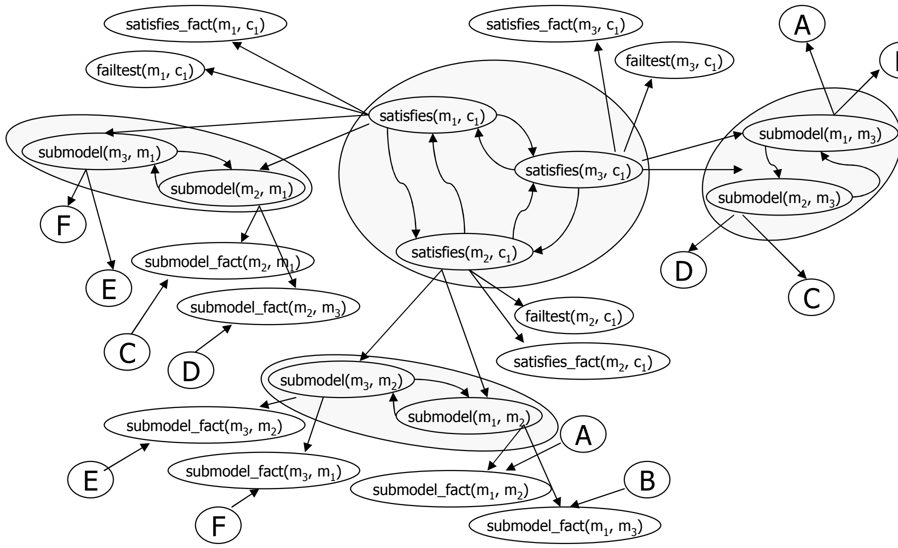

Fig. 5. A part of the atom dependency graph for the *EOLC* program E_1. Note the cliques involving the recursive predicates 'satisfies' and 'submodel'.

m_3)' to 'submodel_fact(m_1, m_3)'. The darker ovals mark the cliques of recursive atoms which when collapsed to super-nodes result in an acyclic graph.

Stratified logic programs also assume restrictions R1 and R2. The algorithm for *EOLC* model computation is similar in spirit to the iterated fixpoint algorithm for computing the perfect (or well-founded or stable) model of stratified logic programs [23]. We however have two different kinds of negation, explicit and default, in *EOLC*, and we perform the fixpoint iteration over a 4-valued interpretation; in addition to the rule ranks and rule priorities.

Let ρ be a stratification of the *EOLC* program E, s.t., all `overrides` atoms come before all other atoms. ρ is a reverse topological sort of the modified atom dependency graph of E, where cliques have been replaced by super-nodes. In words, the model, $\langle S, X\rangle$, is computed iteratively by a series of partial models. The i^{th} step considers p_i, the i^{th} element in the stratification. If p_i is not a super-node, the i^{th} iteration involves a contest between rule instances with head p_i at the top of their ordering, and the rule instances with head $\neg p_i$ at the top of their ordering, whose bodies are enabled in the current partial model. Since the atoms are considered in the order of the stratification, when a ground atom p_i is considered, all ground atoms in the bodies of rules with p_i in the head, have already been considered earlier. In case p_i is a super-node, the i^{th} iteration computes a least fix-point of ground instances of the recursively defined predicate entailed in the current partial model.

Theorem 1. *Every EOLC program has a unique maximal model under the restrictions R1–R3.*

Proof. Suppose that *EOLC* program E, has distinct maximal models $\langle S, X\rangle$ and $\langle S', X'\rangle$. Unless $S' = S$ in which case both models are the same (using maximality), both $S'\backslash S$ and $S\backslash S'$ are non-empty. Let A be the set of ground atoms that occur in $Y = S'\backslash S$.
Case 1: let $q \in Y$ be a literal, s.t., no atom in A is reachable from the atom in q in the atom dependency graph of E,

By definition 2 of a model, condition (a) holds on q. Let $\hat{R}$ be the ground rule in condition (a) on q.

Since no atom in A is reachable from the atom in q in the atom dependency graph, $\hat{R}$ is also enabled in $\langle(S'\backslash Y), X''\rangle$, (where X'' is obtained from X' by removing those atoms that no longer satisfy condition (b), definition 2 in $\langle(S'\backslash Y), X'\rangle$). All elements in $pos\_body(\hat{R}) \in S, S'$ occur as the head of some well-grounded rule instances and hence $\hat{R}$ is well-grounded. Since $(S'\backslash Y) \subset S$, $\hat{R}$ is enabled in $\langle S, X\rangle$, and $\nexists$ rule instance, $\hat{R}'$ of a rule R', s.t. $head(\hat{R}') = \neg p$ which is enabled in $\langle S, X\rangle$ and $\hat{R}' \in \mathcal{R'}^T$, s.t., R' is not overridden by some rule instance $\hat{R}''$ of a rule R'' that is enabled in $\langle S, X\rangle$, s.t, $head(\hat{R}'') = q$ and $\hat{R}'' \in \mathcal{R''}^T$. Hence $q \in S$, meaning that $\langle S, X\rangle$ is not maximal, which is a contradiction.
Case 2: $\nexists q \in Y$, s.t., no atom in A is reachable from the atom in q.

In this case, $\forall y \in Y$, there is no well-grounded rule instance that does not include some other $y' \in Y$ in *pos_body*. By definition of well-grounded-ness these rule instances cannot be well-grounded. This implies that there are no well-grounded rule instances with head $y \in Y$ that are enabled in $\langle S'\backslash Y, X''\rangle$ where X'' is some subset of X' obtained by removing those literals that no longer satisfy condition (b) in the definition of a model (definition 2). Hence we get a contradiction in that $\langle S', X'\rangle$ cannot be a model. □

Theorem 2. *(Tractability of inferencing) The maximal model of an EOLC program, E, is computed in time* $O(n^{2(v+1)})$ *where* $n = size(E)$ *and* v *is the upper bound on the number of variables that occur in rules in E.*

The algorithm for computing the maximal model of an *EOLC* program along with complete proofs and detailed complexity analysis may be found in [18].

5 Relationship to other Approaches

This section compares *EOLC* with some other relevant non-monotonic formalisms, especially those with explicit rule prioritization. A more elaborate discussion is in [18].
Courteous Logic Programs: These were introduced recently as a form of default reasoning useful for intelligent agents [8]. Courteous logic program have unique *answer sets* in the acyclic Datalog-restricted domain. A courteous logic program can, however, be translated into an equivalent *EOLC* program. The inferences drawn from a courteous logic program may be drawn from the translated *EOLC* program. Courteous logic programs do not support recursive concepts, constraints or the ability to select a desirable solution as the `rank` construct does in *EOLC*.

Logic Programming without Negation as Failure (LPwNF): LPwNF (Kakas et al. [24]) is an alternative way of default reasoning with an explicit priority relation among rules. LPwNF, however, considers conflicts between single rules while *EOLC* considers sets of rules with complementary heads at the same time. In the example in figure 5, intuitively it should be possible to infer $mammal(platypus)$ since for every reason for $\neg mammal(platypus)$ there is a stronger reason for $mammal(platypus)$. This intuitive inference is allowed in *EOLC* but not in LPwNF. Brewka's form of prioritized extended logic programs is also unable to handle this example [25]. The well-founded semantics given by Brewka target cyclic dependencies including those involving negation (which *EOLC* does not do).

```
lays_eggs(platypus).
has_fur(platypus).
monotreme(platypus).
has_bill(platypus).
⟨l1⟩ mammal(X): monotreme(X).
⟨l2⟩ mammal(X): has_fur(X).
⟨l3⟩ ¬mammal(X): lays_eggs(X).
⟨l3⟩ ¬mammal(X): has_bill(X).
overrides(l1, l3).
overrides(l2, l4).
```

Fig. 6. Platypus.

LPwNF also does not support the selection of desirable solutions, as the `rank` construct allows one to (refer to figure 3). To our knowledge, there is no other non-monotonic reasoning formalism with priorities that supports such a feature.

Prioritized Logic Programs: (Sakama et al. [6]) It supports an explicit prioritization among literals in an extended logic program to define a preference relation among answer sets. Prioritized logic programs, however, do not support features of *EOLC* regarding selection of a desirable solution using `rank` and also do not have the computational simplicity of *EOLC*.

Other Approaches for Non-monotonic reasoning: Grosof shows that courteous logic programs are more expressive than default inheritance systems such as Touretzky [26], e.g., support for negation, multiple conditions in rule bodies, which also applies to *EOLC*. Zhang and Foo [27] introduce priorities but their formalization considers individual rules and is unable to handle situations like figure 5, and also the features of *EOLC* regarding constraints and `rank`. Their approach is however more general in that it supports arbitrary recursion as in extended logic programs. Brewka's prioritized default logic [28] allows variables to be quantified in both rule heads and bodies. In this sense it is more expressive that *EOLC*, but it does not support features of *EOLC* regarding constraints and `rank` and also does not have the computational simplicity of *EOLC*.

Courteous logic programs, LPwNF, prioritized logic programs, prioritized default logic and other approaches discussed above, do not distinguish the cases when a literal a is overspecified ($\top$) and unknown ($\bot$) and hence cannot deal with situations as in figure 3.

6 Conclusions and Future Work

This paper presents a formalism for commonsense reasoning, *EOLC*, that gives a non-monotonic rule framework extending prioritized defaults and supports recursively defined rules in a four-valued logic. *EOLC* also supports constraints along with an aggregation operator `rank`. We present a simple declarative semantics and an efficient inferencing algorithm for *EOLC* and contrasted *EOLC* with other non-monotonic formalisms. *EOLC* is thus a simple formalism computationally and conceptually, but rich expressively. We are currently integrating a Java implementation of the *EOLC* model computation algorithm into a knowledge management system using the Protégé tool from Stanford. We are also working on abduction with explicit rule priorities in the context of *EOLC*.

Acknowledgements: We thank Frank Pfenning at Carnegie Mellon University for helpful discussions.

References

1. McCarthy, J.: Programs with common sense. In: Proceedings of the Teddington Conference on the Mechanization of Thought Processes, London, Her Majesty's Stationary Office (1959) 75–91
2. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9**(3/4) (1991) 365–386
3. Clark, K.L.: Negation as failure. In: Logic and Data Bases. (1977) 293–322
4. McCarthy, J.: Circumscription: a form of non-monotonic reasoning. Artificial Intelligence **13** (1980) 27–39
5. Reiter, R.: On reasoning by default. In: TINLAP-2: Proceedings of the theoretical issues in natural language processing-2. (1978) 210–218
6. Sakama, C., Inoue, K.: Prioritized logic programming and its application to commonsense reasoning. Artif. Intell. **123**(1-2) (2000) 185–222
7. Antoniou, G., Billington, D., Maher, M.: Sceptical logic programming based default reasoning - Defeasible logic rehabilitated. Formalization of Commonsense Reasoning (1998)
8. Grosof, B.: Courteous logic programs: Prioritized conflict handling for rules. IBM Research Report RC20836 (1997)
9. Baral, C., Gelfond, M.: Logic programming and knowledge representation. Journal of Logic Programming **19/20** (1994) 73–148
10. Blair, H.A., Subrahmanian, V.S.: Paraconsistent logic programming. Theor. Comput. Sci. **68**(2) (1989) 135–154
11. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. **33**(3) (2001) 374–425
12. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. Journal of Logic Programming **19/20** (1994) 503–581

13. Marriott, K., Stuckey, P.J.: Semantics of constraint logic programs with optimization. ACM Lett. Program. Lang. Syst. **2**(1-4) (1993) 197–212
14. Fages, F.: From constraint minimization to goal optimization in CLP languages. In: Principles and Practice of Constraint Programming. (1996) 537–538
15. Govindarajan, K., Jayaraman, B., Mantha, S.: Preference logic programming. In: International Conference on Logic Programming. (1995) 731–745
16. Kumar, R., Krogh, B.H., Feiler, P.: An ontology-based approach to heterogeneous verification of embedded control systems. Hybrid Systems Computation and Control (HSCC) (2005) 370–385
17. Matiyasevich., Y.: Enumerable sets are diophantine. In: English translation in Soviet Mathematics. Doklady. Volume 11. (1970)
18. Kumar, R., Tiwari, A., Krogh, B.: EOLC: A Knowledge Representation Framework for Commonsense Reasoning. Technical report, ECE Department, Carnegie Mellon Univ. (2006)
19. Dung, P., Kowalski, R., Toni, F.: Argumentation-theoretic proof procedures for default reasoning. Technical report, Imperial College, London, UK (1997)
20. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. (1988) 89–148
21. Gelfond, M.: The stable model semantics for logic programming (1988)
22. Ross, K.A., Srivastava, D., Stuckey, P.J., Sudarshan, S.: Foundations of aggregation constraints. In: Principles and Practice of Constraint Programming. (1994) 193–204
23. van Gelder, A., Ross, K., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3) (1991) 620–650
24. Dimopoulos, Y., Kakas, A.: Logic programming without negation as failure. In: 5th. International Symposium on Logic Programming. (1995) 369–384
25. Brewka, G.: Well-founded semantics for extended logic programs with dynamic preferences. Journal of Artificial Intelligence Research **4** (1996) 19–36
26. Touretzky, D.S.: The mathematics of inheritance systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1986)
27. Zhang, Y., Foo, N.Y.: Answer sets for prioritized logic programs. In: International Logic Programming Symposium. (1997) 69–83
28. Brewka, G.: Reasoning about priorities in default logic. In: AAAI National Conference on Artificial Intelligence. Volume 2. (1994) 940–945