In F. Bry, N. Henze and J. Maluszynski (Eds.), Principles and Practice of Semantic Web Reasoning
Proc. of the 1st International Workshop, PPSW3 '03. Springer-Verlag LNCS 2901, 2003

# Web Rules Need Two Kinds of Negation

Gerd Wagner

Eindhoven University of Technology, Faculty of Technology Management,
G.Wagner@tm.tue.nl,
WWW home page: http://tmitwww.tm.tue.nl/staff/gwagner

**Abstract.** In natural language, and in some knowledge representation systems, such as extended logic programs, there are two kinds of negation: a weak negation expressing non-truth, and a strong negation expressing explicit falsity. In this paper I argue that, like in several basic computational languages, such as OCL and SQL, two kinds of negation are also needed in the Semantic Web.

## 1 Introduction

In [Wag91], I have argued that a database, as a knowledge representation system, needs two kinds of negation to be able to deal with partial information. The present paper is an attempt to make the same point for the Semantic Web.

Computational forms of negation are used in imperative programming languages (such as *Java*), in database query languages (such as *SQL*), in modeling languages (such as *UML/OCL*), in production rule systems (such as *CLIPS* and *Jess*) and in logic programming languages (such as *Prolog*). In imperative programming languages, negation may occur in the condition expression of a conditional branching statement. In database query languages, negation may occur in at least two forms: as a *not* operator in selection conditions, and in the form of the relational algebra *difference* operator (corresponding to the SQL *EXCEPT* operator). In modeling languages, negation occurs in constraint statements. E.g., in OCL, there are several forms of negation: in addition to the *not* operator in selection conditions also the *reject* and the *isEmpty* operators are used to express a negation. In production rule systems, and in logic programming languages, a negation operator *not* typically occurs only in the condition part of a rule with the operational semantics of *negation-as-failure* which can be understood as classical negation under the preferential semantics of stable models.

We show in section 2 that negation in all these computational systems is, from a logical point of view, not a clean concept, but combines classical (Boolean) negation with negation-as-failure and the strong negation of three-valued logic (also called *Kleene negation*). In any case, however, it seems to be essential for all major computational systems to provide different forms of negation. Consequently, we may conclude (by common sense induction) that the Semantic Web also needs these different forms of negation.

In natural language, there are (at least) two kinds of negation: a weak negation expressing non-truth (in the sense of "she doesn't like snow" or "he doesn't trust you"), and a strong negation expressing explicit falsity (in the sense of "she dislikes snow" or "he distrusts you"). Notice that the classical logic law of the excluded middle holds only for the weak negation (either "she likes snow" or "she doesn't like snow"), but not for the strong negation: it does not hold that "he trusts you" or "he distrusts you"; he may be neutral and neither trust nor distrust you.

A number of knowledge representation formalisms and systems, discussed in section 4, follow this distinction between weak and strong negation in natural language. However, many of them do not come with a model-theoretic semantics in the style of classical logic. Instead, an inference operation, that may be viewed as a kind of proof-theoretic semantics, is proposed.

Classical (two-valued) logic cannot account for two kinds of negation because two-valued (Boolean) truth functions do not allow to define more than one negation. The simplest generalization of classical logic that is able to account for two kinds of negation is *partial logic* giving up the classical bivalence principle and subsuming a number of 3-valued and 4-valued logics. For instance, in 3-valued logic with truth values $\{f, u, t\}$ standing for *false*, *undetermined* (also called *unknown* or *undefined*) and *true*, weak negation (denoted by $\sim$) and strong negation (denoted by $\neg$) have the following truth tables:

| $p$ | $\sim p$ |
|---|---|
| $t$ | $f$ |
| $u$ | $t$ |
| $f$ | $t$ |

| $p$ | $\neg p$ |
|---|---|
| $t$ | $f$ |
| $u$ | $u$ |
| $f$ | $t$ |

Notice the difference between weak and strong negation in 3-valued logic: if a sentence evaluates to $u$ in a model, then its weak negation evaluates to $t$, while its strong negation evaluates to $u$ in this model. Partial logics allow for *truth-value gaps* created by partial predicates to which the law of the excluded middle does not apply.

However, even in classical logic, where all predicates are total, we may distinguish between predicates that are completely represented in a database (or knowledge base) and those that are not. The classification if a predicate is completely represented or not is up to the owner of the database: the owner must know for which predicates she has complete information and for which she does not. Clearly, in the case of a completely represented predicate, negation-as-failure amounts to classical negation, and the underlying completeness assumption is also called *Closed-World Assumption*. In the case of an incompletely represented predicate, negation-as-failure only reflects non-provability, but does not allow to infer the classical negation. Unfortunately, neither CLIPS/Jess nor Prolog support this distinction between 'closed' and 'open' predicates.

Open (incompletely represented total) predicates must not be confused with partial predicates that have truth-value gaps. The law of the excluded middle, $p \vee \neg p$, applies to open predicates but not to partial predicates.

For being able to make all these distinctions and to understand their logical semantics, we have to choose partial logic as the underlying logical framework. Partial logic allows to formally distinguish between falsity and non-truth by means of strong and weak negation. In the case of a total predicate, such as being an odd number, both negations collapse:

$$\sim odd(x) \text{ iff } \neg odd(x),$$

or in other words, the non-truth of the atomic sentence $odd(x)$ amounts to its falsity. In the case of a partial predicate, such as *likes*, we only have the relationship that the strong negation implies the weak negation:

$$\sim likes(she,snow) \text{ if } \neg likes(she,snow),$$

but not conversely. Also, while the double negation form '$\neg \sim$' collapses (according to partial logic, see [Wag98]), the double negation form '$\sim \neg$' does not collapse: not disliking snow does not amount to liking snow. Classical logic can be viewed as the degenerate case of partial logic when all predicates are total.

## 2 Negative Information, Closed Predicates and Two Kinds of Negation in the Semantic Web

We claim that, like in the cases of UML/OCL, SQL and extended logic programs, also for the Semantic Web one needs two kinds of negation. This applies in particular to RDF/RDFS, OWL and RuleML.

### 2.1 Expressing Negative Information in RDF

The *FIPA RDF Content Language Specification* (see www.fipa.org) that specifies how RDF can be used as a message content language in the communication acts of FIPA-compliant agents proposes a method how to express negated RDF facts to *'express belief or disbelief of a statement'*. For this purpose an RDF statement (expressed as a 'subject-predicate-object' triple corresponding to *objectID-attribute-value*) is annotated by a truth value *true* or *false* in a `<fipa:belief>` element as in the following example:

```
<fipa:Proposition>
  <rdf:subject>RDF Semantics</rdf:subject>
  <rdf:predicate rdf:resource="http://description.org/schema#author"/>
  <rdf:object>Ora Lassila</rdf:object/>
  <fipa:belief>false</fipa:belief>
</fipa:Proposition>
```

This example expresses the negated sentence

*Ora Lassila is not the author of 'RDF Semantics'.*

It shows that there is a need to extend the current syntax of RDF, so as to be able to express negative information.

### 2.2 Closed Predicates and Negation-as-Failure in RDF/S

As opposed to the predicate 'is the author of', there are also predicates for which there is no need to express negative information because the available positive information about them is complete and, consequently, the negative information is simply the complement of the positive information.

For instance, the W3C has complete information about all official W3C documents and their normative status (http://www.w3.org/TR/ is the official list of W3C publications); consequently, the predicate *is an official W3C document* should be declared as closed in the W3C knowledge base (making a 'local' completeness assumption).[1] This consideration calls for a suitable extension of RDFS in order to allow making such declarations for specific predicates.

For sentences formed with closed predicates it is natural to use negation-as-failure for establishing their falsity (anything not listed on that page cannot be a W3C recommendation). So, a query language for RDF should include some form of negation-as-failure.

### 2.3 Default Rules in RuleML and N3

The RuleML standardization initiative has been started in August 2000 with the goal of establishing an open, vendor neutral XML-based rule language standard. The official website of the RuleML initiative is `www.ruleml.org`.

The current 'official' version of RuleML (in July 2003) has the version number 0.84. In [BTW01], the rationale behind RuleML 0.8 and some future extensions of it is discussed, while [Wag02] provides a general discussion of the issues of rule markup languages.

An example of a derivation rule involving strong negation (for making sure that something is definitely not the case) and negation-as-failure (for expressing a default condition) is the following:

> *A car is available for rental if it is not assigned to any rental order, does not require service and is not scheduled for a maintenance check.*

This rule could be marked up in RuleML as shown in Figure 1. Strong negation is expressed by `<neg>` while negation-as-failure is expressed by `<naf>`. Notice that it is important to apply `<neg>`, and not `<naf>`, to `requiresService` in order to make sure, by requiring explicit negative information, that the car in question does not require service (the car rental company may be liable for any consequences/damages caused by a failure of this check).

However, the last condition of the rule, expressed with `<naf>`, is a default condition requiring only that there is no information about any assignment of the car in question.

In N3, one can test for what a formula does not say, with `log:notIncludes`. In the following example (taken from [BL]), we have a rule stating that if the specification for a car doesn't say what color it is, then it is black:

[1] This example is due to Sandro Hawke, see [DDM].

```
this log:forAll :car. { :car.auto:specification log:notIncludes {:car
auto:color []}}
=> {:car auto:color auto:black}.
```

In this rule, the `log:notIncludes` operator expresses a negation-as-failure in a similar way as the `isEmpty` operator of OCL and the `IS NULL` operator of SQL.

## 3 Negations in UML/OCL, SQL, CLIPS/Jess and Prolog

UML/OCL, SQL, CLIPS/Jess and Prolog may be viewed as the paradigm-setting languages for *modeling*, *databases*, *production rules* and (logical) *derivation rules*. We discuss each of them in some more detail.

### 3.1 Negation in UML/OCL

The Unified Modeling Language (UML) may be viewed as the paradigm-setting language for software and information systems modeling. In the UML, negation occurs in Object Constraint Language (OCL) statements. There are several forms of negation in OCL: in addition to the *not* operator in selection conditions also the *reject* and the *isEmpty* operators are used to express a negation. OCL allows partially defined expressions and is based on a 3-valued logic where the third truth value, denoted by $\perp$, is called *undefined*.

The above rule for rental cars defines the derived Boolean-valued attribute `isAvailable` of the class `RentalCar` by means of an association `isAssignedTo` between cars and rental orders and the stored Boolean-valued attributes `requiresService` and `isSchedForMaint`. All these concepts are shown in the UML class diagram in Figure 2. Notice that `requiresService` is defined as an optional attribute (that need not always have a value). This reflects the fact that whenever a rental car is returned by a customer, it is not known if it requires service until its technical state is checked. Only then this attribute obtains a value true or false.[2] As opposed to `requiresService`, `isSchedForMaint` is defined as a mandatory attribute that must always have a value, reflecting the fact that the car rental company always knows if a car is or is not scheduled for a maintenance check.

Since the Object Constraint Language (OCL) of UML does not allow to define derivation rules, we have to express the definition of the derived attribute `isAvailable` by means of an OCL invariant statement:

```
context RentalCar inv:
  RentalOrder->isEmpty
    and not requiresService
    and not isSchedForMaint
  implies isAvailable
```

[2] Notice that optional, i.e. partial, attributes in the UML correspond to SQL table columns admitting null values.

This integrity constraint states that for a specific rental car whenever there is no rental order associated with it, and it does not require service and is not scheduled for maintenance, then it has to be available for a new rental. It involves three forms of negation:

1. the first one, `RentalOrder->isEmpty`, expresses the negation-as-failure *there is no information that the car is assigned to any rental order*;
2. the second one, `not requiresService`, is strong negation; and
3. the third one, `not isSchedForMaint`, is classical negation.

We discuss each of them in more detail.

**The *not* Operator** The negation in `not isSchedForMaint` is classical negation, since `isSchedForMaint` is defined as a mandatory Boolean-valued attribute. However, the negation in `not requiresService` is strong negation, since `requiresService` is defined as an optional Boolean-valued attribute such that the truth value of the corresponding statement is *unknown* whenever the value of this attribute is `NULL`. Thus, viewing Boolean-valued attributes as *predicates*, we may say that UML allows for both (closed) total and partial predicates, such that *not* denotes classical (Boolean) negation when applied to a total predicate and strong (Kleene) negation when applied to a partial predicate.

**The *isEmpty* Operator** The negation that is implicitly expressed by `RentalOrder->isEmpty` is negation-as-failure, since it evaluates to true whenever there is no information about any associated rental order. Notice, however, that having no information about any associated rental order does logically not imply that there is no associated rental order. Only in conjunction with a completeness assumption (either for the entire database or at least for the predicate concerned) can we draw this conclusion.

In summary, we have three kinds of negation in OCL: classical negation, strong negation and negation-as-failure.

### 3.2 Negation in SQL

In SQL, negation may occur in various forms: as a `NOT` operator or as an `IS NULL` operator in selection conditions, or in the form of the `EXCEPT` table operator (corresponding to the relational algebra *difference* operator). SQL may be viewed as the paradigm-setting language for databases. It supports null values and incomplete predicates (whose truth-value may be *unknown*), and is based on a 3-valued logic with the truth values *true*, *unknown* and *false*, where `NOT` corresponds to strong negation [MS02].

The following SQL table definition implements the class `RentalCar` from the UML class diagram of Figure 2.

```
CREATE TABLE  RentalCar(
CarID             CHAR(20) NOT NULL,
requiresService   BOOLEAN,
isSchedForMaint   BOOLEAN NOT NULL,
isAvailable       BOOLEAN,
isAssignedTo      INTEGER REFERENCES RentalOrder
)
```

Notice that `isSchedForMaint` is defined as a mandatory ('not null') Boolean-valued column, whereas `requiresService` is defined as an optional Boolean-valued column and `isAssignedTo` as an optional reference to a rental order. Table 1 contains a sample population of the RentalCar table.

| *CarID* | *requiresService* | *isSchedForMaint* | *isAvailable* | *isAssignedTo* |
|---|---|---|---|---|
| 23010 | false | false | false | 1032779 |
| 23011 | false | false | true | NULL |
| 23785 | NULL | false | NULL | NULL |
| 30180 | true | true | false | NULL |

**Table 1.** A sample population of the RentalCar table.

In SQL databases, a *view* defines a derived table by means of a query. For instance, the derived table of available cars is defined as the view

```
CREATE VIEW AvailableCar( CarID)
  SELECT CarID FROM RentalCar
  WHERE isAssignedTo IS NULL
    AND NOT requiresService
    AND NOT isSchedForMaint
```

The `SELECT` statement in this view contains three negations:

1. the first one, `isAssignedTo IS NULL`, expresses the negation-as-failure stating that *there is no information that the car is assigned to any rental order*;
2. the second one, `NOT requiresService`, is strong negation; and
3. the third one, `NOT isSchedForMaint`, is classical negation.

We discuss each of them in more detail.

**The *not* Operator** When applied to a complete predicate, SQL's *not* expresses classical negation, but when applied to an incomplete predicate, it expresses strong negation because SQL evaluates logical expressions using 3-valued truth functions, including the truth table for ¬ presented in the introduction.

When we ask the query *'which cars do* not *require service?'* against the database state shown in Table 1, using the SQL statement

```
SELECT CarID FROM RentalCar
WHERE NOT requiresService
```

we actually use strong negation because `requiresService` is an incomplete predicate (admitting NULL values). Thus, the resulting answer set would be $\{23010, 23011\}$. That SQL's *not* behaves like strong negation when applied to an incomplete predicate can be demonstrated by asking the query *'which cars require service or do* not *require service?'*:

```
SELECT CarID FROM RentalCar
WHERE requiresService OR NOT requiresService
```

leading to the result set $\{23010, 23011, 30180\}$. If *not* would be classical negation in this query, then, according to the law of the excluded middle, the answer should be the set of all cars from table `RentalCar`, that is $\{23010, 23011, 23785, 30180\}$. However, SQL's answer set includes only those cars for which the `requiresService` attribute has the value *true* or *false*, but not those for which it is NULL.

**The *IS NULL* Operator** When we ask, however, 'which cars are *not* assigned to any rental order?' using the SQL statement

```
SELECT CarID FROM RentalCar
WHERE isAssignedTo IS NULL
```

leading to the result set $\{23011, 23785, 30180\}$, we use negation-as-failure because without a completeness assumption, the `isAssignedTo IS NULL` condition does not imply that there is really no associated rental order, but only that there is no information about anyone.

**The *EXCEPT* Operator** Also, SQL's `EXCEPT` operator corresponds to Prolog's negation-as-failure *not*: a Prolog query expression "give me all objects x such that 'p(x) and not q(x)'" corresponds to the SQL expression 'P EXCEPT Q' where P and Q denote the tables that represent the extensions of the predicates p and q.

### 3.3 Negation in CLIPS/Jess and Prolog

CLIPS/Jess and Prolog may be viewed as the paradigm-setting languages for *production rules* and (computational logic) *derivation rules*. Both languages have been quite successful in the Artificial Intelligence research community and have been used for many AI software projects. However, both languages also have difficulties to reach out into, and integrate with, mainstream computer science and live rather in a niche. Moreover, while Prolog has a strong theoretical foundation (in computational logic), CLIPS/Jess and the entire production rule paradigm lack any such foundation and do not have a formal semantics. This problem is partly due to the fact that in production rules, the semantic categories of

events and conditions in the left-hand-side, and of actions and effects in the right-hand-side, of a rule are mixed up.

While derivation rules have an if-*Condition*-then-*Conclusion* format, production rules have an if-*Condition*-then-*Action* format. To determine which rules are applicable in a given system state, conditions are evaluated against a fact base that is typically maintained in main memory.

In Prolog, the rule for available cars is defined by means of the following two rules:

```
availableCar(X) :-
  rentalCar(X),
  not requiresService(X),
  not isSchedForMaint(X),
  not isAssignedToSomeRental(X).

isAssignedToSomeRental(X) :-
  isAssignedTo(X,Y).
```

The second of these rules is needed to define the auxiliary predicate `isAssignedToSomeRental` because Prolog does not provide an existential quantifier in rule conditions for expressing a formula like $\neg\exists y(p(x, y))$. Although they include the possibility of using the nonmonotonic negation-as-failure operator, Prolog rules and deductive database rules (including SQL views) have a purely declarative semantics in terms of their intended models (in the sense of classical logic model theory). For rules without negation, there is exactly one intended model: the unique *minimal* model. The intended models of a set of rules with negation(-as-failure) are its *stable* models.

Production rules do not explicitly refer to events, but events can be simulated by asserting corresponding objects into working memory. A derivation rule can be simulated by a production rule of the form if-*Condition*-then-assert-*Conclusion* using the special action *assert* that changes the state of a production rule system by adding a new fact to the set of available facts.

The production rule system *Jess*, developed by Ernest Friedman-Hill at Sandia National Laboratories, is a Java successor of the classical LISP-based production rule system *CLIPS*. Jess supports the development of rule-based systems which can be tightly coupled to code written in the Java programming language. As in LISP, all code in Jess (control structures, assignments, procedure calls) takes the form of a function call. Conditions are formed with conjunction, disjunction and negation-as-failure. Actions consist of function calls, including the assertion of new facts and the retraction of existing facts.

In Jess, the rule for available cars is defined as

```
(defrule availableCar
  (and (RentalCar ?x)
       (not (requiresService ?x))
       (not (isSchedForMaint ?x))
       (not (isAssignedToSomeRental ?x)))
  =>
  (assert (availableCar ?x))
```

Both Prolog and Jess allow negation to be used only in the body (or condition) of a rule, and not in its head (in Jess, the head of a rule represents an action, so negation wouldn't make sense here, anyway), nor in facts. So, unlike in SQL, where a Boolean-valued attribute can have the value *false* as distinct from `NULL` corresponding to *unknown*, there is no possibility to represent and process explicit negative information. For instance, the negative fact that *the car with CarID=23010 does not require service*, expressed by the attribute-value pair *23010.requiresService=false* in Table 1, cannot be represented in Jess and Prolog. In both languages, *not* expresses negation-as-failure implementing classical negation in the case of complete predicates subject to a completeness (or 'Closed-World') assumption .

This shortcoming has led to the extension of normal logic programs by adding a negation for expressing explicit negative information, as proposed independently in [GL90,GL91], and in [PW90,Wag91].

## 4 Two Kinds of Negation in Knowledge Representation

A number of knowledge representation formalisms and systems follow the distinction between weak and strong negation in natural language which is also implicit in SQL. We mention just two of them:

- Logic programs with two kinds of negation (called *extended logic programs* in [GL90]).
- The IBM business rule system CommonRules (described in [Gro97,GLC99]) that is based on the formalism of extended logic programs.

Using two kinds of negation in derivation rules has been proposed independently in [GL90] and [Wag91]. Unfortunately, and confusingly, several different names and several different semantics have been proposed by different authors for these two negations. Strong negation has been called 'classical negation' and 'explicit negation', while negation-as-failure has been renamed into 'implicit negation' and 'default negation'. In particular, the name 'classical negation' is confusing because (the real) classical negation satisfies the law of the excluded middle while the 'classical negation' in extended logic programs does not. Apparently, the reason for choosing the name 'classical negation' is of a psychological nature: one would like to have classical negation, or at least some approximation of it. But that's exactly what partial logic is able to offer: for complete predicates, both strong negation and weak negation collapse into classical negation.

Unlike for logical theories in standard logics, the semantics of knowledge bases in knowledge representation formalisms is not based on all models of a knowledge base but solely on the set of all intended models. E.g., for relational databases the intended models are the 'minimal' ones in the intuitive sense of minimal information content. However, a satisfactory definition of minimally informative models is not possible in classical logic, but only in partial logics. Among all partial models of a KB the minimal ones are those that make a minimal number of atomic sentences true or false. This definition does not work

for classical models where a sentence is false iff it is not true. So, classical models allow only an asymmetric definition of minimality: one may define that among all classical models of a KB the minimal ones are those that make a minimal number of atomic sentences true. However, this definition does not adequately capture the intuitive notion of minimal information content, since both the truth and the falsity of a sentence should count as information.

For a KB consisting of derivation rules with negation-as-failure, minimal model semantics is not adequate, because it does not account for the directedness of such rules. This is easy to see. Consider the knowledge base $\{p \leftarrow \text{not } q\}$. This KB has two minimal models: $\{p\}$ and $\{q\}$, but only $\{p\}$ is an intended model.

The model-theoretic semantics of derivation rules with negation-as-failure (e.g. in normal and extended logic programs) is based on the concept of stable (generated) classical models (see [GL88,HW97]). Under the preferential semantics of stable (generated) models, classical negation corresponds to negation-as-failure, or, in other words, negation-as-failure implements classical negation under the preferential semantics of stable (generated) models.

There is a kind of proof-theoretic semantics for normal logic programs, called *wellfounded semantics*, originally proposed by [vG88]. It should be rather considered an inference operation (or a proof theory) which is sound but incomplete with respect to stable model semantics.

The model-theoretic semantics of derivation rules with negation-as-failure and strong negation (e.g. in extended logic programs) is based on the concept of stable generated partial models (see [HJW99]). Under the preferential semantics of stable generated partial models, weak negation corresponds to negation-as-failure, or, in other words, negation-as-failure implements weak negation when applied to an incomplete predicate, and it implements classical negation when applied to a complete predicate.

Another model-theoretic semantics for extended logic programs, which is elegant but more complicated (since based on possible worlds), is the *equilibrium* semantics of [Pea99]. Other proposed semantics, such as the *answer set semantics* of [GL90,GL91] or the WFSX semantics of [PA92], are not model-theoretic and less general (they do not allow for arbitrary formulas in the body and head of a rule).

In the next section, we present the logical formalism needed to explain two kinds of negation.

## 5 Partial Logics with Two Kinds of Negation and Two Kinds of Predicates

This section is based on [HJW99,Wag98].

A function-free[3] partial logic *signature* $\sigma = \langle Pred, TPred, Const\rangle$ consists of a set of predicate symbols *Pred*, the designation of a set of total predicate symbols $TPred \subseteq Pred$, and a set of constant symbols *Const*.

### 5.1 Partial Models

We restrict our considerations to Herbrand interpretations since they capture the *Unique Name Assumption* which is fundamental in the semantics of databases and logic programming.

**Definition 1 (Interpretation)** *Let $\sigma = \langle Pred, TPred, Const\rangle$ be a signature. A partial Herbrand $\sigma$-interpretation $\mathcal{I}$ consists of:*

1. *A set $U_\mathcal{I}$, called* universe *or* domain *of $\mathcal{I}$, which is equal to the set of constant symbols, $U_\mathcal{I} = Const$;*
2. *an assignment $\mathcal{I}(c) = c$ to every constant symbol $c \in Const$;*
3. *an assignment of a pair of relations $\mathcal{I}_t(p), \mathcal{I}_f(p)$ to every predicate symbol $p \in Pred$ such that*
$$\mathcal{I}_t(p) \cup \mathcal{I}_f(p) \subseteq U_\mathcal{I}^{a(p)},$$
*and in the special case of a total predicate $p \in TPred$,*
$$\mathcal{I}_t(p) \cup \mathcal{I}_f(p) = U_\mathcal{I}^{a(p)},$$
*where $a(p)$ denotes the arity of $p$.*

In the sequel we also simply say 'interpretation' ('satisfaction', 'model', 'entailment') instead of 'partial Herbrand interpretation' ('partial Herbrand satisfaction', 'partial Herbrand model', 'partial Herbrand entailment').

The class of all $\sigma$-interpretations is denoted by $\boldsymbol{I}_4(\sigma)$. We define the classes of *coherent*, of *total*, and of total coherent (or *2-valued*) interpretations by

$$\begin{aligned}
\boldsymbol{I}_c(\sigma) &= \{\mathcal{I} \in \boldsymbol{I}_4(\sigma) \mid \mathcal{I}_t(p) \cap \mathcal{I}_f(p) = \emptyset \text{ for all } p \in Pred\}\\
\boldsymbol{I}_t(\sigma) &= \{\mathcal{I} \in \boldsymbol{I}_4(\sigma) \mid \mathcal{I}_t(p) \cup \mathcal{I}_f(p) = U_\mathcal{I}^{a(p)} \text{ for all } p \in Pred\}\\
\boldsymbol{I}_2(\sigma) &= \boldsymbol{I}_c(\sigma) \cap \boldsymbol{I}_t(\sigma)
\end{aligned}$$

The model relation $\models$ between a Herbrand interpretation and a sentence is defined inductively as follows.

**Definition 2 (Satisfaction)**

$$\begin{aligned}
\mathcal{I} \models p(c_1, \ldots, c_m) &\iff \langle c_1, \ldots, c_m\rangle \in \mathcal{I}_t(p)\\
\mathcal{I} \models \neg p(c_1, \ldots, c_m) &\iff \langle c_1, \ldots, c_m\rangle \in \mathcal{I}_f(p)\\
\mathcal{I} \models \sim F &\iff \mathcal{I} \not\models F\\
\mathcal{I} \models F \wedge G &\iff \mathcal{I} \models F \ \ \&\ \ \mathcal{I} \models G\\
\mathcal{I} \models F \vee G &\iff \mathcal{I} \models F \ \ \textit{or}\ \ \mathcal{I} \models G\\
\mathcal{I} \models \exists x F(x) &\iff \mathcal{I} \models F(c) \textit{ for some } c \in Const\\
\mathcal{I} \models \forall x F(x) &\iff \mathcal{I} \models F(c) \textit{ for all } c \in Const
\end{aligned}$$

[3] For simplicity, we exclude function symbols from the languages under consideration, i.e. we do not consider functional terms but only variables and constants; signatures without function symbols lead to a finite Herbrand universe.

All other cases of compound formulas are handled by the following DeMorgan and double negation rewrite rules:

$$\begin{array}{cc} \neg(F \wedge G) \longrightarrow \neg F \vee \neg G & \neg(F \vee G) \longrightarrow \neg F \wedge \neg G \\ \neg\exists x F(x) \longrightarrow \forall x \neg F(x) & \neg\forall x F(x) \longrightarrow \exists x \neg F(x) \\ \neg\neg F \longrightarrow F & \neg \sim F \longrightarrow F \end{array}$$

in the sense that for every rewrite rule $LHS \longrightarrow RHS$, we define

$$\mathcal{I} \models LHS \iff \mathcal{I} \models RHS$$

$\mathrm{Mod}_*$ denotes the model operator associated with the system $\langle L(\sigma), \boldsymbol{I}_*, \models\rangle$, and $\models_*$ denotes the corresponding entailment relation, for $* = 4, c, t, 2$, i.e.

$$X \models_* F \quad \text{iff} \quad \mathrm{Mod}_*(X) \subseteq \mathrm{Mod}_*(\{F\})$$

**Observation 1** *If only two-valued models are admitted, weak and strong negation collapse:*

$$\neg F \equiv_2 \sim F$$

### 5.2 Classical Logic as a Special Case of Partial Logic

Obviously, the entailment relation $\models_2$ corresponds to entailment in classical logic. The most natural way to arrive at classical logic from proper partial logic is to assume that all predicates are total: $TPred = Pred$. Under this assumption, the two entailment relations $\models_c$ and $\models_2$ of partial logic collapse.

*Claim.* If $TPred = Pred$, then $\models_c = \models_2$.

### 5.3 Total Predicates and the Closed-World Assumption

In general, three kinds of predicates can be distinguished. The first distinction, proposed in [Koe66], reflects the fact that many predicates (especially in empirical domains) have truth value gaps: neither $p(c)$ nor $\neg p(c)$ has to be the case for specific instances of such *partial* predicates, like, e.g., color attributes which can in some cases not be determined because of vagueness.

Other predicates, e.g. from legal or theoretical domains, are *total*, and we then have, for instance, $m(S) \vee \neg m(S)$ and

$$\mathrm{prime}(2^{77} - 1) \vee \neg\mathrm{prime}(2^{77} - 1)$$

stating that Sophia is either married or unmarried, and that $2^{77} - 1$ is either a prime or a non-prime number. Only total predicates can be completely represented in a knowledge base. Therefore, only total predicates can be subject to a completeness assumption. For simplicity, a predicate is called *closed* whenever it is completely represented, otherwise it is called *open*.

For distinguishing between closed, open total and partial predicates, the schema of a knowledge base has to specify a set $Pred = \{p_1, \ldots, p_n\}$ of predicates (or table schemas), a set $TPred \subseteq Pred$ of total predicates, and a set $CPred \subseteq TPred$ of closed predicates.

**Definition 3 (Completeness Assumption)** *For a knowledge base $Y$ over a schema specifying a set of closed predicates* CPred*, we obtain the following additional inference rule for drawing negative conclusions,*

$$Y \vdash \neg p(c) \quad \textit{if} \quad p \in \mathit{CPred} \ \ \& \ \ Y \vdash {\sim}p(c)$$

The completeness assumption, in a less general form, was originally proposed in [Rei78], under the name *Closed-World Assumption (CWA)*. Our form of the CWA relates explicit with default-implicit falsity, i.e. strong with weak negation. It states that an atomic sentence formed with a closed predicate is false if it is false by default, or, in other words, its strong negation holds if its weak negation does. It can also be expressed by means of the *completion* $\mathit{Compl}(Y)$ of a knowledge base $Y$ with respect to the set of closed predicates *CPred*:

$$\mathit{Compl}(Y) \ = \ \mathsf{Upd}(Y, \{\neg p(c) \mid p \in \mathit{CPred} \ \ \& \ \ Y \vdash {\sim}p(c)\})$$

A sentence $F$ is inferable from $Y$ if it can be derived from the *tertium-non-datur*-closure of $\mathit{Comp}(Y)$:

$$Y \vdash F :\Longleftrightarrow \mathsf{Upd}(\mathit{Compl}(Y),\ \{p(\boldsymbol{c}) \vee \neg p(\boldsymbol{c}) \mid p \in \mathit{TPred} - \mathit{CPred}\}) \vdash F$$

Notice that in definite knowledge bases (not admitting disjunctions), it is not possible to declare total predicates that are open. Therefore, in definite knowledge systems, $\mathit{TPred} = \mathit{CPred}$.

**Observation 2** *For a knowledge base $Y$, it holds that*

1. *for any total predicate $p \in \mathit{TPred}$, and any constant (tuple) $c$, the resp. instance of the* tertium non datur *holds: $Y \vdash p(c) \vee \neg p(c)$;*
2. *if $q \in \mathit{CPred}$, then $Y$ does not contain any indefinite information about $q$, i.e. $Y \vdash q(c)$, or $Y \vdash \neg q(c)$.*

### 5.4 Reasoning with Three Kinds of Predicates

Only certain total predicates can be completely represented in a KB. These closed predicates are subject to the completeness assumption. For example, the KB of a city may know all residents of the city, i.e. the completeness assumption holds for *resident*, but it does not have complete information of every resident whether (s)he is married or not because (s)he might have married in another city and this information is not available. Consequently, the completeness assumption does not apply to *married* in this KB.

The completeness assumption helps to reduce disjunctive complexity which is exponential in the number of open total predicates: if $n$ is the number of unknown ground atoms which can be formed by means of predicates declared as total but open, then the knowledge base contains $2^n$ possible state descriptions.

We illustrate these distinctions with an example. Let $m, r, s, l$ denote the predicates *married*, *resident*, *smoker* and *is_looking_at*, and let $M, P, A$ stand for the individuals *Mary*, *Peter* and *Ann*. Let

$$Y = \{\{m(M),\ r(M),\ s(M),\ \neg m(A),\ \neg s(A),\ l(M,P),\ l(P,A)\}\}$$

be a knowledge base over a schema declaring the predicates $m$ and $r$ to be total, and the predicate $r$ to be closed. The interesting queries we can ask $Y$ and the resp. answers are:

1. Does a married person look at an unmarried one? Yes, but $Y$ does not know who, either Mary at Peter, or Peter at Ann. Formally, it holds that

$$Y \vdash \exists x \exists y (l(x,y) \wedge m(x) \wedge \neg m(y))$$

   but there is no definite answer to this query, only an indefinite answer may be obtained:

$$\mathsf{Ans}(Y,\, l(x,y) \wedge r(x) \wedge \neg r(y)) = \{\{\langle M, P\rangle,\, \langle P, A\rangle\}\})$$

2. Does a resident look at a non-resident ? Yes, Mary at Peter.

$$\mathsf{Ans}(Y,\, l(x,y) \wedge r(x) \wedge \neg r(y)) = \{\{\langle M, P\rangle\}\}$$

   since $Y \vdash \neg r(P)$ if $Y \vdash {\sim} r(P)$.
3. Does a smoker look at a nonsmoker? No. $Y$ is completely ignorant about Peter being a smoker or not: neither is he a smoker, nor is he a nonsmoker, nor is he a smoker or nonsmoker (as a partial predicate, $s$ may have a truth value gap for this instance):

$$\mathsf{Ans}(Y,\, l(x,y) \wedge s(x) \wedge \neg s(y)) = \emptyset$$

## 6 Adding Two Kinds of Negation to RDF, OWL and RuleML

It would be plausible to declare the predicate *is the author of* as open total, since we cannot assume that the RDF knowledge base in question has complete information about all authors of all books.

### 6.1 Closed Predicates and Negation-as-Failure in RDF/S

The predicate *is an official W3C document* should be declared as closed. This consideration calls for a suitable extension of RDFS in order to allow making such declarations for all predicates.

Having negated RDF facts, and open and closed predicates, suggests to use both strong negation as well as negation-as-failure in RDF queries.

## 7 Conclusion

Like many other computational systems and formalisms, also the Semantic Web would benefit from distinguishing between open and closed predicates using both strong negation and negation-as-failure. We have shown that partial logic is the logic of these two kinds of negation. Consequently, it would be important to generalize RDF and OWL from their current classical logic version to a suitable partial logic version and to combine them with RuleML.

## References


[BL] Tim Berners-Lee. Reaching out onto the web. web document. http://www.w3.org/2000/10/swap/doc/Reach.

[BTW01] Harold Boley, Said Tabet, and Gerd Wagner. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In *Proc. Semantic Web Working Symposium (SWWS'01)*. Stanford University, July/August 2001.

[DDM] Stefan Decker, Mike Dean, and Deborah McGuinness. Requirements and use cases for a semantic web rule language. web document. http://www.isi.edu/ stefan/rules/20030325/.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proc. of ICLP*, pages 1070–1080. MIT Press, 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proc. of Int. Conf. on Logic Programming*. MIT Press, 1990.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[GLC99] B.N. Grosof, Y. Labrou, and Hoi Y. Chan. A declarative approach to business rules in contracts: Courteous logic programs in XML. In *Proc. 1st ACM Conference on Electronic Commerce (EC99)*, Denver, Colorado, USA, November 1999.

[Gro97] B.N. Grosof. Prioritized conflict handling for logic programs. In Jan Maluszynski, editor, *Proc. of the Int. Symposium on Logic Programming (ILPS-97)*. MIT Press, 1997.

[HJW99] H. Herre, J. Jaspars, and G. Wagner. Partial logics with two kinds of negation as a foundation of knowledge-based reasoning. In D.M. Gabbay and H. Wansing, editors, *What Is Negation?* Oxford University Press, 1999.

[HW97] H. Herre and G. Wagner. Stable models are generated by a stable chain. *J. of Logic Programming*, 30(2):165–177, 1997.

[Koe66] S. Koerner. *Experience and Theory*. Kegan Paul, London, 1966.

[MS02] Jim Melton and Alan R. Simon. *SQL:1999*. Morgan Kaufmann, San Francisco, CA, 2002.

[PA92] Luís Moniz Pereira and José Júlio Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 102–106. Wiley, 1992.

[Pea99] D. Pearce. Stable inference as intuitionistic validity. *Journal of Logic Programming*, 38:79–91, 1999.

[PW90] D. Pearce and G. Wagner. Reasoning with negative information I – strong negation in logic programs. In M. Kusch L. Haaparanta and I. Niiniluoto, editors, *Language, Knowledge and Intentionality*. Acta Philosophica Fennica 49, 1990.

[Rei78] R. Reiter. On closed-world databases. In J. Minker and H. Gallaire, editors, *Logic and Databases*. Plenum Press, 1978.

[vG88] A. van Gelder. Negation as failure using tight derivations for general logic programs. In *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, San Mateo (CA), 1988.

[Wag91] G. Wagner. A database needs two kinds of negation. In B. Thalheim and H.-D. Gerhardt, editors, *Proc. of the 3rd. Symp. on Mathematical Fundamentals of Database and Knowledge Base Systems*, volume 495 of *Lecture Notes in Computer Science*, pages 357–371. Springer-Verlag, 1991.

[Wag98] G. Wagner. *Foundations of Knowledge Systems – with Applications to Databases and Agents*, volume 13 of *Advances in Database Systems*. Kluwer Academic Publishers, 1998. See http://www.inf.fu-berlin.de/∼wagnerg/ks.html.

[Wag02] Gerd Wagner. How to design a general rule markup language? In *Proceedings of the First Workshop on XML Technologies for the Semantic Web (XSW2002)*, Lecture Notes in Informatics, Berlin, June 2002. Gesellschaft fr Informatik. invited paper.

```
<imp>
  <_head>
    <atom>
      <_opr>isAvailable</_opr>
      <var>Car</var>
    </atom>
  </_head>
  <_body>
     <and>
      <atom>
        <_opr>RentalCar</_opr>
        <var>Car</var>
      </atom>
      <neg>
        <atom>
          <_opr>requiresService</_opr>
          <var>Car</var>
        </atom>
      </neg>
      <naf>
        <atom>
          <_opr>isSchedForMaint</_opr>
          <var>Car</var>
        </atom>
      </naf>
      <naf>
        <atom>
          <_opr>isAssToRentalOrder</_opr>
          <var>Car</var>
        </atom>
      </naf>
    </and>
  </_body>
</imp>
```

**Fig. 1.** The rule for available cars marked up in RuleML.

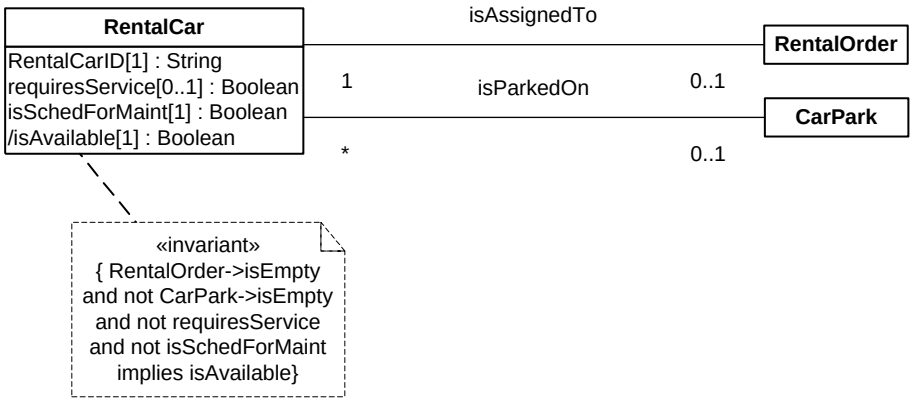


**Fig. 2.** This UML class diagram shows two classes, `RentalCar` and `RentalOrder`, and the functional association `isAssignedTo` between them. The Boolean-valued attribute `isAvailable` is a derived attribute whose definition is expressed by the attached OCL constraint.