The Modular Web Framework
User's manual

Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou

1. Introduction

The Modular Web Framework (MWeb) allows the use and combination of several rulebases in a principled way, supporting safe uses of non-monotonic negation in scoped closed and open world assumptions in logic rules for Semantic Web applications.

Currently it supports experimentally RDF Schema ontologies and is integrated with the state-of-the-art inference engines XSB and Smodels.
Support of RIF and Integration with OWL-2 is foreseen in the near future.

MWeb's main features are

- Safe support of monotonic (strong) and non-monotonic (weak or naf) negations under full control of rulebase providers and consumers;
- Immediate specification of scoped open and closed world assumptions independently of the existing or shared rulebases;
- Full-fledged theory and results guaranteeing monotonicity under broad conditions;
- Independent interpretations of shared knowledge via explicit or implicit export and import directives;
- Adoption and transparent use of main semantics for logic rule bases (well-founded and answer set based);
- Detection of knowledge dependent on contradictions;
- Separate interface and rulebase module code;
- Embellished Prolog-like syntax for simplifying rulebase development;
- Modular and independent compilation of modules with dynamic loading;
- Compilation into XSB prolog modules with tabling to guarantee termination and efficiency;
- Seamless integration with XSB Prolog system with no overhead.

2. Copyright and Contacts

The MWeb software has been developed by

Carlos Viegas Damásio(cd@di.fct.unl.pt)
Centro de Inteligência Artificial (CENTRIA)
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre
2829-516 Caparica
Portugal

The MWeb framework is a joint work of Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damásio with contributions from Gerd Wagner. More information can be found at MWeb's web page http://centria.di.fct.unl.pt/~cd/mweb.

3. Download and installation

The MWeb framework assumes that XSB Prolog version 3.2 or later is properly installed with Smodels support. The XSB system can be obtained from http://xsb.sourceforge.net.
The Smodels system is distributed with XSB, but users can obtain further information at http://www.tcs.hut.fi/Software/smodels/ .

After that, download the tgz archive from http://centria.di.fct.unl.pt/~cd/mweb and unpack it in a directory of your choice (let's denote it by `USERDIR/mweb`).

For simpler use we suggest the following line to be added to your `xsbrc.P` file, usually located in `~/.xsb` directory (do not forget the quotes):

```
:- assert(library_directory('USERDIR/mweb/source')).
```

Compile the MWeb framework with the command `xsb mweb`. You should obtain in the output something like (ignore any warning messages that might exist) the following sequence of text (parts have been removed):

```
$ xsb mweb
[xsb_configuration loaded]
[sysinitrc loaded]
[siteinitrc loaded]
[xsbrc loaded]
[Compiling /Users/carlosdamasio/Documents/Codigo/mweb/mweb]
% Specialising partially instantiated calls to mwebExecuteGoal/4
% Specialising partially instantiated calls to mwebCompileGoal/4
% Specialising partially instantiated calls to ord_union/3
[mweb compiled, cpu time used: 0.0730 seconds]
[xasp loaded]
[sm_clause_store loaded]
[Compiling /Users/carlosdamasio/Documents/Codigo/mweb/mwebParser]
% Specialising partially instantiated calls to declare_prefixes/3
% Specialising partially instantiated calls to process_mweb_scope/5
% Specialising partially instantiated calls to process_mweb_predicate/4
% Specialising partially instantiated calls to process_mweb_defines_predicates/8
% Specialising partially instantiated calls to process_mweb_uses_mode/5
% Specialising partially instantiated calls to process_mweb_atom/6
% Specialising partially instantiated calls to process_curie/3
% Specialising partially instantiated calls to process_strong_negation/2
% Specialising partially instantiated calls to process_slots/3
% Specialising partially instantiated calls to process_and_slots/3
% Specialising partially instantiated calls to expand_mweb_atom/6
[Module mwebParser compiled, cpu time used: 0.1400 seconds]
 …
[Compiling /Users/carlosdamasio/Documents/Codigo/mweb/mwebCompiler]
% Specialising partially instantiated calls to mwebLocalPredicateName/3
% Specialising partially instantiated calls to mwebEncodeLiteral/4
% Specialising partially instantiated calls to mwebCompileListUses/3
% Specialising partially instantiated calls to mwebCompileOneUses/7
% Specialising partially instantiated calls to mwebExportDeclarations/5
% Specialising partially instantiated calls to mwebCompileLiteral/4
% Specialising partially instantiated calls to mwebPredicateName/5
% Specialising partially instantiated calls to mwebDualPrologPredicate/1
[Module mwebCompiler compiled, cpu time used: 0.0900 seconds]
[mwebCompiler loaded]
[Compiling /Users/carlosdamasio/Documents/Codigo/mweb/mwebRuntime]
[Module mwebRuntime compiled, cpu time used: 0.0040 seconds]
[mwebRuntime loaded]

End XSB (cputime 0.32 secs, elapsetime 0.38 secs)
```

Check that the installation is fine by cding to `USERDIR/mweb/examples/security` and consulting file `testSecurity`. The output should be similar to:

```
| ?- [testSecurity].
[Compiling ./testSecurity]
[testSecurity compiled, cpu time used: 0.0130 seconds]
[testSecurity loaded]
[mweb loaded]
[xasp loaded]
[sm_clause_store loaded]
[mwebParser loaded]
[fancyProlog loaded]
[iri loaded]
[iriparse loaded]
[utilities loaded]
[builtins loaded]
[mwebCompiler loaded]
[mwebRuntime loaded]
[Compiling ./http%3A%2F%2Fgeography%2Eint]
[Preprocessing ./http%3A%2F%2Fgeography%2Eint.pl]
[Module http%3A%2F%2Fgeography%2Eint compiled, cpu time used: 0.0150 seconds]
[Compiling ./http%3A%2F%2Feuropa%2Eeu]
[Preprocessing ./http%3A%2F%2Feuropa%2Eeu.pl]
++Warning[XSB]: [Compiler] ./http%3A%2F%2Feuropa%2Eeu : Unused symbol $rbnaf/1
++Warning[XSB]: [Compiler] ./http%3A%2F%2Feuropa%2Eeu : Unused symbol @/2
++Warning[XSB]: [Compiler] ./http%3A%2F%2Feuropa%2Eeu : Unused symbol
tneg_http://geography.int#Country/2
++Warning[XSB]: [Compiler] ./http%3A%2F%2Feuropa%2Eeu : Unused symbol usermod/0
++Warning[XSB]: [Compiler] ./http%3A%2F%2Feuropa%2Eeu : Unused symbol
tpos_http://geography.int#Country/2
++Warning[XSB]: [Compiler] ./http%3A%2F%2Feuropa%2Eeu : Unused symbol
tupos_http://geography.int#Country/2
++Warning[XSB]: [Compiler] ./http%3A%2F%2Feuropa%2Eeu : Unused symbol @@/2
++Warning[XSB]: [Compiler] ./http%3A%2F%2Feuropa%2Eeu : Unused symbol
tuneg_http://geography.int#Country/2
[Module http%3A%2F%2Feuropa%2Eeu compiled, cpu time used: 0.0160 seconds]
[Compiling ./http%3A%2F%2Fsecurity%2Eint]
[Preprocessing ./http%3A%2F%2Fsecurity%2Eint.pl]
[Module http%3A%2F%2Fsecurity%2Eint compiled, cpu time used: 0.0250 seconds]
[Compiling ./http%3A%2F%2Fgov%2Ecountry]
[Preprocessing ./http%3A%2F%2Fgov%2Ecountry.pl]
++Warning[XSB]: [Compiler] ./http%3A%2F%2Fgov%2Ecountry : Unused symbol @@/2
[Module http%3A%2F%2Fgov%2Ecountry compiled, cpu time used: 0.0500 seconds]

yes
```

Ignore the warnings and execute the query `load, test`, obtaining exactly the following output:

```
| ?- load, test.
MWebWFS solutions:
Anne

MWebAS solutions:
Anne
Boris

yes
| ?-
```

You are now ready to start learning to use the MWeb system.

4. Overview and syntax of the MWeb framework

MWeb rulebases are defined by an interface and a logic program. By convention, the current implementation of MWeb assumes that these are specified in files with extensions '.mw' and '.rb', respectively.

The MWeb rulebase interface specifies the predicates defined and visible to the outside world (exported predicates) as well as the used (imported) predicates. The logic program of the rulebase defines by means of logic rules the meaning of the predicates defined in the rulebase. By design principles all defined and used predicated must be declared.

The current implementation of the MWeb framework specifies rulebases in syntax very similar to Prolog with a slight user-friendly differences. It is expected that other input formats to be supported in the future (e.g. RIF, N3, Jena, Prolog, etc…).

The MWeb Interface

The syntax of the MWeb interface is given by the following pattern, where terminals are quoted in '**bold**', and non-terminals in *italic*. Notice that the several language constructs are aligned with RIF syntax in order to foster future integration. All statements end with a dot, and terminals can be separated with whitespace. Comments can be put inside pairs **/* */** or be line comments starting with **%**.

Table 1. MWeb interface file constructs

```
 ':-' 'rulebase' IRIRef '.'
(':-' 'base' IRIRef  '.')?
(':-' 'prefix' Name '=' IRIRef (',' Name '=' IRIRef)* '.' )*
(':-' 'import(' FileName ',' ' interface' ')' '.')*
(':-' 'vocabulary' Const (',' Const)* '.')*
(':-' DefinesDecl)*
(':-' UsesDecl)*
```

The MWeb interface starts with a mandatory rulebase declaration providing its unique identifier. For maintaining compatibility with Prolog, it is allowed to use an IRI reference as module identifier instead of restricting exclusively to Absolute IRIs. For uniformity, IRI references should be put inside quotes.
An optional Base IRIRef can be specified to resolve Compact URIs for the default namespace. The optional declaration of other namespace prefixes and its association with IRIRefs follow immediately. Notice that several prefix declarations can occur sequentially in the same interface file.
The contents of other MWeb interface files can be in-place inserted via the import declaration. For the time being the file should be in the machine's file system and cannot be loaded from the Web. Do not forget the parenthesis.
The users can declare domain constants by resorting to the optional vocabulary declaration. Notice that all constants found in the corresponding rulebase logic program are automatically added to the rulebase vocabulary without any user intervention. However, for some applications it might be necessary to declare extra constants not found in the logic program.

The MWeb interface file continues with a sequence of **defines** declarations and **uses** declarations, which roughly correspond to export and import declarations respectively. These declarations are detailed on Table 2 and the syntax is given by an EBNF grammar.

Table 2. Grammar of defines and uses declarations

```
DefinesDecl ::= 'defines' ScopeDecl RegimePred
                ['wrt' 'context' PredicateInd ] [ 'visible' 'to' RulebaseList ] '.'

UsesDecl::= 'uses' RegimePred [ 'from' RulebaseList ] '.'
       |  'import (' ('definite' | 'normal') PredicateList 'from' RulebaseIRI ')' '.'

ScopeDecl ::= 'global' | 'local' | 'internal'

RegimePred ::= ('definite' | 'open' | 'closed' | 'normal' ) PredicateList

PredicateList ::= PredicateInd ( ',' PredicateList )*
PredicateInd ::=  PosPredicateInd | NegPredicateInd

NegPredicateInd ::= 'neg' PosPredicateInd
PosPredicateInd ::= CURIE '/' NATURAL | Atom |
                    'class(' CURIE ')' | 'property(' CURIE ')'

RulebaseList ::= RulebaseIRI ( , RulebaseIRI )*
RulebaseIRI ::= IRIRef

CURIE ::= PREFIX ':' LOCALNAME | IRIRef

PREFIX ::= Name
LOCALNAME ::= Name
IRIRef ::= Name | '\'' PROLOGCHAR* '\''
Const ::= PROLOGCONST | '"' PROLOGCHAR* '"' '^^' CURIE
```

The defines declaration starts with the scope specifying the span of a predicate. Next, the regime (reasoning mode) should be mandatorily specified. Several predicates can be specified with a single defines declaration. Afterwards, an optional context predicate can be given for the objective and closed modes (ignored in the other cases). Finally, visibility can be controlled by providing an explicit list of rulebases which can use the predicates; if absent the predicates defined can be used by any rulebase. The defines declaration is synthesized on Table 3.

The meaning of class and property constructs is explained in the ERDF section of this document where syntax is expanded to cater for RIF frames. However, it should be noticed that it is expected a unary context predicate for classes and a binary one for properties (more discussion on the examples section).

The previous EBNF grammar is dependent on the following symbols: PROLOGCHAR, PROLOGCONST, *Name*, and *Atom*. The terminal PROLOGCHAR matches Prolog characters, briefly ASCII character. The terminal PROLOGCONST matches any atomic symbol defined by Prolog syntax. *Name* is an alphanumeric string of literals starting in lower or upper case, while *Atom* matches an optionally prefixed logical atom with arguments. Here users can use names starting with upper case but need to prefix variables with a question mark '?' symbol. These are detailed in Table 6.

Table 3. Summary syntax and meaning of the defines declaration

`defines`	*ScopeDecl*	
	`global`	The predicates can be defined in several rulebases.
	`local`	The predicates can be defined only in a single rulebase.
	`internal`	The predicates are invisible to other rulebases.
	RegimePred	
	`definite`	The rules for the predicates are monotonic. Non-weakly negated conclusions are monotonic. Only strong negation can be used in the defining rules and normal predicates are forbidden. Similar to Horn programs but with strong negation.
	`open`	The rules for the predicates are monotonic with open world assumption. Non-weakly negated conclusions are monotonic. Only strong negation can be used in the defining rules and normal predicates are forbidden. Similar to Horn programs but with strong negation allowed with open world assumption.
	`closed`	The rules for the predicates are associated with a positive or negative closed world assumption, depending on the keyword **neg** in *PredicateList*. Conclusions are non-monotonic. Only strong negation can be used in the defining rules and normal predicates are forbidden. Similar to Horn programs but with strong negation allowed with closed world assumption.
	`normal`	Arbitrary rules with monotonic (strong - neg) and non-monotonic (weak - naf) negation. Conclusions are non monotonic. Corresponds to extended logic programs.
	PredicateList	Specifies the comma separated list of the defined predicates. Each predicate can have the keywork **neg** before it. This keyword is used in regime closed to indicate that the predicate is negatively closed; otherwise it is ignored.
	Prefix:Local/N	Defines a predicate with N arguments with name constructed by appending the namespace IRI with Local. An IRIref should be obtained, unless the reserved prefix **atom** is used.
	Prefix:Local(?A1,...,?An)	Defines a predicate with **n** arguments with name constructed by appending the namespace IRI with Local. A valid IRIref should be obtained. The list of arguments is explicitly provided and can be used to bind arguments with the context predicate.
	Local/N	The same as before but it is appended to the default base IRI to construct the name. For compatibility with Prolog if there is no base IRI then Local is used as name. A valid IRI reference should be obtained.
	Local(?A1,...,?An)	
`wrt context`	*PredicateInd*	Specifies an optional context predicate for the open or closed world assumptions of objective and closed regimes. The arity (number of arguments) of this predicate must be the same as any defined predicate in the list. The binding of variables of the context predicate with the defined predicates can be implicitly left to right (using arity) or explicit (using explicit variables). If absent, the open and closed world assumptions are done with respect to the vocabulary of any loaded rulebase.
`visible to`	*RulebaseList*	Limits the use of the non-internal predicates to the rulebases in the specified comma separated list. If absent, the non-internal predicates can be used by any rulebase.

The uses declaration is simpler to use and just requires the expressing of the import regime and the provider rulebases. If the rulebase list is absent then the predicate is imported from

any loaded rulebase defining the predicate in either global or local scope. The regime of the used predicate is combined with the regime of the defined declaration in order to obtain the regime followed at query time. The rules for obtaining the regime are condensed on Table 4, roughly meaning that it is used the least regime of the use and defines declaration where regimes are totally ordered by **definite** < **open** < **closed** < **normal**. The X marks an error condition which can be disabled with a call to `mwebSetErrorLevel(none)`.

Table 4. Combination of regime modes

uses \ defines	d	o	c	n
d	d	d	d	X
o	d	o	o	X
c	d	o	c	X
n	d	o	c	n

It is also provided a declaration that can be used to import XSB prolog predicates under normal regime; XSB builtins can also be used by importing from the appropriate module (or usermod for inline predicates). The use of parenthesis is mandatory (mark the extra space).

Finally, notice that a predicate can be defined and used in the same MWeb interface file by allowing users to redefine a particular implementation of a predicate defined elsewhere. In particular, if defined in internal scope, even local predicates can be used internally and redefined in other rulebases.

The MWeb Logic Program

The syntax of MWeb rulebases is very similar to Prolog, with some single exceptions in order to approximate the syntax to the one used by RIF. The summary of the syntax is collected on Table 5 and Table 6. The syntax will be extended subsequently to handle RIF syntax, in particular RIF frames, equality, class membership and subclass relation (see Section 5 for more details). Additionally, the MWeb logic program allows the definition of internal predicates as well as inline importing of other MWeb logic programs. The general syntax is covered by the grammar on Table 5.

Table 5. Syntax of MWeb logic programs

```
MWebLogicProgram ::= Statement*
Statement ::= ':-' Import | ':-' DefinesInternal | Rule
Import ::= 'import(' FileName ',' ' rulebase' ')' '.'
DefinesInternal ::= 'defines internal' RegimePred ['wrt' 'context' PredicateInd ] '.'
```

The definition of internal predicates is explained in the previous section. The import directive is recursively substituted by the contents of *Filename*. The syntax of rules is presented and explained next.

Table 6. Syntax of MWeb logic rules

Name	Syntax	Comment

Rule	*Implies* | *Fact*	
Implies	*Head* ':-' *Body* '.'	A rule if Body then Head. Head must be an atom or the strong negation of an atom, while Body is an arbitrary formula.
Fact	*Head* '.'	Abbreviation for a rule with true body.
And	*Conj1* ',' … ',' *ConjN*	A conjunction (separator is comma)
Or	*Disj1* ';' … ';' *DisjN*	A disjunction (separator is semi colon)
Qualified	*Default* '@' *RulebaseIRI*	A call to *Default* literal in *RulebaseIRI*
Default	'naf' *Objective* | '~' *Objective*	Weakly (nonmonotonic) negated *Objective* literal. Naf is the preferred syntax.
Objective	'neg' *Atom* | '-' *Atom*	Strongly negated *Atom* (monotonic). Neg is the preferred syntax. '-' should be avoided.
Atom	(Prefix ':')? *Name* ('(' *Term1* ',' … ',' *TermN* ')')?	An optionally prefixed *Name* with an optional list of term arguments.
Term	*Const* | *Var* | (Prefix ':')? *Name* ('(' *Term1* ',' … ',' *TermN* ')')?	A term is either a constant, a variable or a complex term.
Const	A XSB Prolog constant or a typed literal '"' PROLOGCHAR* '"' '^^' *CURIE*. Currently no syntax checking of typed literals is performed.	
Var	'?' *Name* | '?' _[*Name*]	Differently from Prolog, a variable must be prefixed with a question mark. The name of the variable can be an arbitrary name (i.e. start with lower case, or even be a Prolog atom). ?_ is an anonymous variable.
Name	A name can be an identifier starting with a letter (lower or upper-case!) separated with underscores or an arbitrary UTF-8 string inside quotes.	

The special predicate `Domain( +ListOfVars )` can be used in bodies of rules to instantiate unbound variables with all the vocabulary available from loaded rulebases. This is particularly useful when using rules with weak negations in order to instantiate the corresponding variables; otherwise, floundering problems may occur.

The MWebWFS and MWebAS semantics

Two semantics are defined for the MWeb framework, and selected at query time. The MWebWFS is a semantics based on Paraconsistent Well-fouded Semantics with Explicit negation. Ground queries can be evaluated in polynomial time on the size of the instantiated rulebase and existence of a single model is guaranteed. Users can check consistency of a query by checking if L and naf L are simultaneously true. If so, the query is itself contradictory (and thus neg L is also true) or dependent on contradiction. This semantics is the best tractable approximation to MWebAS currently known. No consistency checks are done for this semantics.

The MWebAS semantics is the semantics based on Answer Set semantics, and consequently has the benefits and disadvantages of the latter. Consistency checks are done automatically for this semantics and some rulebases may not have a model. In particular, in face of contradictions everything is entailed from the program, and users are informed of inconsistency. This semantics is computationally more expensive (entailment is coNP-complete) and thus more expressive (see the examples).

5. Experimental Support of RIF and (E)RDF Schema

The MWeb implementation is currently being extended to support both a dialect as specialization of the Rule Interchange Format (RIF) and the Extended Resource Description Framework Schema (ERDF). In order to maintain compatibility with RIF and generality, all properties and classes are assumed to be normal, allowing for the use of weak negation in the bodies. However, as expected, monotonicity is not guaranteed.

The implementation of these extensions is done through MWeb rulebases where the interface and meaning can be entirely defined within MWeb. This demonstrates the simplicity, usefulness, and generality of our approach. A preliminary implementation is available for testing and already interesting behaviour can be obtained.

RIF support

The supported dialect implements fully the semantics of classification (member and subclass), frames, and partially equality. Regarding connectives, the usual binary boolean connectives conjunction and disjunction, as well as strong and weak negations are supported (under Paraconsistent Well-founded Semantics with Explicit Negation and Answer Set Semantics). No disjunction is allowed in the heads of rules.

Syntactically, the MWeb parser supports RIF frames (molecules) of the form

?O.[?A1 ->> ?V1, ..., ?An ->> ?Vn]

which internally are translated into a conjunction (mark the order of arguments)

'->'(?A1,?O,?V1), ..., '->'(?An,?O,?Vn),

The other binary RIF predicates '=' (equality), '#' (member, Isa), and '##' (subclass) have the exact syntax of RIF. The semantics of these predicates are provided by the following MWeb rulebase:

RIF's MWeb interface (rif.mw)	RIF's MWeb logic program (rif.rb)
:- rulebase 'http://www.w3.org/2007/rif'. :- prefix rif='http://www.w3.org/2007/rif#'. :- defines internal normal name:'='/2. :- defines internal normal name:'#'/2. :- defines internal normal name:'##'/2. :- defines internal normal name:'->'/3. :- defines global normal class(mw:Vocabulary).	% RIF member relation extended to handle strong negation ?O # ?CL :- ?O # ?SCL, ?SCL ## ?CL. neg ?O # ?SCL :- neg ?O # ?CL, ?SCL ## ?CL. % RIF subclass relation. No rule needed for strong negation ?C1 ## ?C3 :- ?C1 ## ?C2, ?C2 ## ?C3. % RIF partial equality theory. ?T = ?T :- ?T # mw:Vocabulary. ?T1 = ?T2 :- ?T2 = ?T1. ?T1 = ?T3 :- ?T1 = ?T2, ?T2 = ?T3. neg ?T1 = ?T2 :- neg ?T2 = ?T1. neg ?T1 = ?T3 :- ?T1 = ?T2, neg ?T2 = ?T3. % RIF frames obtained by equality reasoning. ?O.[?P ->> ?V] :- ?O1.[?P ->> ?V], ?O = ?O1. ?O.[?P ->> ?V] :- ?O.[?P1 ->> ?V], ?P = ?P1. ?O.[?P ->> ?V] :- ?O.[?P ->> ?V1], ?V = ?V1.

RIF predicates are declared internal in order to prevent disclosure of unintended information. Users can then make visible instances of particular predicates or classes by declaring them local or internal. The special class **mw:Vocabulary** collects all the defined vocabulary implicitly or explicitly in vocabulary declarations.

Class inheritance is captured by the first rule in file 'rif.rb'. Mark the need to have a rule for taking care of the case where it is known that something does not belong to the extension of some class (second rule). For subclass relation it is only required transitivity; transitivity does not have a dual negative rule. Frames do not have any special meaning, thus no rules are needed to capture them.

The most difficult predicate to capture is equality. Our approach is designed to handle correctly equality among constants; no recursion on arguments is performed. For instance, if some user states that a = b and b =c the consequence c = a is obtained but not f(a) = f(c). The OWL 2/RL rules of equality can be easily captured.

Users wishing to use RIF should import the files 'rif.mw' and 'rif.rb' into their MWeb interface and logic program files, respectively (see example in the next subsection).

ERDF support

The MWeb's implementation of the Extended Resource Description Framework builds on the RIF frame constructs and is made in three stages: *rdf* support, *rdfs* support, and finaly *erdf* support. All the three can be used independently by importing the corresponding files.

The semantics of the combination is the one adopted by RIF and specified in the W3C Candidate Recommendation (RIF RDF and OWL Compatibility). A triple *s p o* is syntactically represented by the RIF frame *s.[p ->> o]*.

In our implementation, the *erdf* rulebase imports the *rdfs* rulebase, and the *rdfs* rulebase imports the *rdf* rulebase. The supporf of RIF primitives is guaranteed by the import of rif by the rdf modular rulebase. For the sake of completeness, we present next the MWeb's implementation of each rulebase for the different profiles (or entailment regimes).

RDF's MWeb interface (rdf.mw)

```
:- rulebase  'http://www.w3.org/1999/02/22-rdf-syntax-ns'.

:- prefix  rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
:- import( 'rif.mw', interface ).

:- defines internal normal class(rdf:Property).
:- defines internal normal class(rdf:XMLLiteral).
:- defines internal normal class(rdf:List).
:- defines internal normal class(rdf:Statement).
:- defines internal normal class(rdf:Seq).
:- defines internal normal class(rdf:Bag).
:- defines internal normal class(rdf:Alt).

:- defines internal normal property(rdf:type).
:- defines internal normal property(rdf:subject).
:- defines internal normal property(rdf:predicate).
:- defines internal normal property(rdf:object).
:- defines internal normal property(rdf:first).
:- defines internal normal property(rdf:rest).
:- defines internal normal property(rdf:value).
```

RDF's MWeb logic program (rdf.rb)

```
:- import( 'rif.rb', rulebase ).

% RDF compatibility with RIF
% makes # and rdf:type equivalent

?X # ?Y :- ?X.[ rdf:type ->> ?Y].
?X.[rdf:type ->> ?Y] :-  ?X # ?Y.

% RDF Entailment rule
?Z.[ rdf:type ->> rdf:Property] :- ?_.[?Z ->> ?_] .
```

```
% RDF Axiomatic triples
rdf:type.[rdf:type->>rdf:Property].
rdf:subject.[rdf:type->>rdf:Property].
rdf:predicate.[rdf:type->>rdf:Property].
rdf:object.[rdf:type ->> rdf:Property].
rdf:first.[rdf:type ->> rdf:Property].
rdf:rest.[rdf:type ->> rdf:Property].
rdf:value.[rdf:type ->> rdf:Property].
rdf:nil.[rdf:type ->> rdf:List].

% Handles container membership properties. They are infinitely many.
% The rule only takes effect if called explicitly with the object grounded.
?X.[rdf:type ->> rdf:Property] :-
        External( name:atom(?X), prolog ),
        External( name:atom_concat( rdf:'_', ?N, ?X ), prolog ),
        External( name:is_number_atom( ?N ), prolog ).
```

The support of RDF entailment is immediate. All the classes and properties defined in RDF are declared internal. The property(CURIE) construct allows to declare RDF properties. In fact, this is syntactic sugar for the RIF frame ?_.[CURIE ->> ?_] resulting in better looking interface files, and expects a binary context predicate. A class declaration `class(CURIE)` is short for `?_ # CURIE`.

By the recommendation governing RIF-RDF compatibility, the predicates '#'/2 and rdf:type should be made equivalent; this is achieved by the first rules. The only rule necessary for RDF entailment states that any predicate of a triple must have type rdf:Property. Then, the axiomatic RDF triples are stated, concluding with the special treatment of RDF container membership properties.

The RDF container membership properties are treated specially since they are infinitely many (rdf:_1, rdf:_2, etc…). The rule only fires if the subject of the triple is bound at query time with an atom; if you need to use these properties please include them on your vocabulary for performing appropriately closed and open world assumptions. Our RIF dialect also supports externally defined terms External(?Term, prolog) to perform Prolog calls, as shown in the implementation of equality; the prefix **name** which is bound to the empty string is essential to avoid resolution against the default base IRI.

RDF Schema entailment is more complex to specify, but immediate. According to RIF-RDF compatibility every RIF subclass instance is also an RDFS subClassOf instance (but not vice-versa). Afterwards, all the RDF schema inference rules and axiomatic triples are encoded; container membership properties are treated as in the implementation of RDF. No other special treatment is necessary expect for XML Literals which are not currently taken care (as well as the other RIF and XML Schema datatypes).

RDFS' MWeb interface (rdfs.mw)

```
:- rulebase 'http://www.w3.org/2000/01/rdf-schema'.
:- prefix rdfs='http://www.w3.org/2000/01/rdf-schema#'.
:- import( 'rdf.mw', interface ).

:- defines internal normal class(rdfs:Resource).
:- defines internal normal class(rdfs:Literal).
:- defines internal normal class(rdfs:Datatype).
:- defines internal normal class(rdfs:Class).
:- defines internal normal class(rdfs:Container),
        class(rdfs:ContainerMembershipProperty).
```

```
:- defines internal normal property(rdfs:domain).
:- defines internal normal property(rdfs:range).
:- defines internal normal property(rdfs:subClassOf).
:- defines internal normal property(rdfs:subPropertyOf).
:- defines internal normal property(rdfs:member).
:- defines internal normal property(rdfs:comment).
:- defines internal normal property(rdfs:seeAlso).
:- defines internal normal property(rdfs:isDefinedBy).
:- defines internal normal property(rdfs:label).
```

RDFS' MWeb logic program (rdfs.rb)

```
:- import( 'rdf.rb', rulebase ).

% RDFS compatibility with RIF requires
% including ## into rdfs:subClassOf
?X.[rdfs:subClassOf ->> ?Y] :- ?X ## ?Y.

% RDFS entailment rules
?Z.[ rdf:type ->> ?Y] :-
      ?X.[rdfs:domain ->> ?Y],  ?Z.[ ?X ->> ?W].
?W.[ rdf:type ->> ?Y] :-
      ?X.[rdfs:range ->> ?Y],  ?Z.[ ?X ->> ?W].

?X.[ rdfs:subClassOf ->> rdfs:Resource] :-
       ?X.[rdf:type ->> rdfs:Class] .

?X.[ rdf:type ->> rdfs:Class ] :- ?X.[ rdfs:subClassOf ->> ?Y ].
?Y.[ rdf:type ->> rdfs:Class ] :- ?X.[ rdfs:subClassOf ->> ?Y ].

?Z.[ rdf:type ->> ?Y] :-
      ?X.[ rdfs:subClassOf ->> ?Y ],  ?Z.[rdf:type ->> ?X].

?X.[ rdfs:subClassOf ->> ?X] :- ?X.[rdf:type ->> rdfs:Class] .
?X.[ rdfs:subClassOf ->> ?Z] :-
      ?X.[ rdfs:subClassOf ->> ?Y],
      ?Y.[ rdfs:subClassOf ->> ?Z] .

?X.[ rdf:type ->> rdf:Property ] :-
     ?X.[ rdfs:subPropertyOf ->> ?Y ].
?Y.[ rdf:type ->> rdf:Property ] :-
     ?X.[ rdfs:subPropertyOf ->> ?Y ].

?Z1.[ ?Y ->> ?Z2] :-
     ?X.[ rdfs:subPropertyOf ->> ?Y ],    ?Z1.[?X ->> ?Z2].

?X.[ rdfs:subPropertyOf ->> ?X] :-
     ?X.[rdf:type ->> rdf:Property] .
?X.[ rdfs:subPropertyOf ->> ?Z] :-
    ?X.[ rdfs:subPropertyOf ->> ?Y],
    ?Y.[ rdfs:subPropertyOf ->> ?Z] .

?X.[ rdfs:subClassOf ->> rdfs:Literal] :-
    ?X.[rdf:type ->> rdfs:Datatype] .
?X.[ rdfs:subPropertyOf ->> rdfs:member] :-
    ?X.[rdf:type ->> rdfs:ContainerMembershipProperty] .
```

```
% RDFS Axiomatic triples
rdf:type.[rdfs:domain ->> rdfs:Resource].
rdfs:domain.[rdfs:domain ->> rdf:Property].
rdfs:range.[rdfs:domain ->> rdf:Property].
rdfs:subPropertyOf.[rdfs:domain ->> rdf:Property].
rdfs:subClassOf.[rdfs:domain ->> rdfs:Class].
rdf:subject.[rdfs:domain ->> rdf:Statement].
rdf:predicate.[rdfs:domain ->> rdf:Statement].
rdf:object.[rdfs:domain ->> rdf:Statement].
rdfs:member.[rdfs:domain ->> rdfs:Resource].
rdf:first.[rdfs:domain ->> rdf:List].
rdf:rest.[rdfs:domain ->> rdf:List].
rdfs:seeAlso.[rdfs:domain ->> rdfs:Resource].
rdfs:isDefinedBy.[rdfs:domain ->> rdfs:Resource].
rdfs:comment.[rdfs:domain ->> rdfs:Resource].
rdfs:label.[rdfs:domain ->> rdfs:Resource].
rdf:value.[rdfs:domain ->> rdfs:Resource].

rdf:type.[rdfs:range ->> rdfs:Class].
rdfs:domain.[rdfs:range ->> rdfs:Class].
rdfs:range.[rdfs:range ->> rdfs:Class].
rdfs:subPropertyOf.[rdfs:range ->> rdf:Property].
rdfs:subClassOf.[rdfs:range ->> rdfs:Class].
rdf:subject.[rdfs:range ->> rdfs:Resource].
rdf:predicate.[rdfs:range ->> rdfs:Resource].
rdf:object.[rdfs:range ->> rdfs:Resource].
rdfs:member.[rdfs:range ->> rdfs:Resource].
rdf:first.[rdfs:range ->> rdfs:Resource].
rdf:rest.[rdfs:range ->> rdf:List].
rdfs:seeAlso.[rdfs:range ->> rdfs:Resource].
rdfs:isDefinedBy.[rdfs:range ->> rdfs:Resource].
rdfs:comment.[rdfs:range ->> rdfs:Literal].
rdfs:label.[rdfs:range ->> rdfs:Literal].
rdf:value.[rdfs:range ->> rdfs:Resource].

rdf:Alt.[rdfs:subClassOf ->> rdfs:Container].
rdf:Bag.[rdfs:subClassOf ->> rdfs:Container].
rdf:Seq.[rdfs:subClassOf ->> rdfs:Container].

?X.[rdf:type ->> rdfs:ContainerMembershipProperty,
rdfs:domain ->> rdfs:Resource, rdfs:range ->> rdfs:Resource] :-
            External( atom:atom(?X), prolog ),
            External( atom:atom_concat( rdf:'_', ?N, ?X ), prolog ),
            External( atom:is_number_atom( ?N ), prolog ).

rdfs:ContainerMembershipProperty.[rdfs:subClassOf ->>
rdf:Property].

rdfs:isDefinedBy.[rdfs:subPropertyOf ->> rdfs:seeAlso].

rdf:XMLLiteral.[rdf:type ->> rdfs:Datatype].
rdf:XMLLiteral.[rdfs:subClassOf ->> rdfs:Literal].
rdfs:Datatype.[rdfs:subClassOf ->> rdfs:Class].
```

The Extended Resource Description Framework introduces the notions of total and closed class, as well as total and closed property, and a mechanism to express complementary properties. Totalness is captured by declaring the class and property having rdf:type erdf:TotalClass and erdf:TotalProperty classes, respectively. For declaring closed classes and properties, the erdf:PositivelyClosedClass, erdf:NegativelyClosedClass, erdf:PositivelyClosedProperty, and erdf:NegativelyClosedProperty can be used, with the expected meaning. The semantics of these ERDF constructs is specified by normal rules, and grounding of variables is made in the code by using the instances of class `mw:Vocabulary`.

ERDF's MWeb interface (erdf.mw)

```
:- rulebase 'http://erdf.org'.
:- prefix erdf='http://erdf.org#'.

:- import( 'rdfs.mw', interface ).

:- defines internal normal class(erdf:TotalClass).
:- defines internal normal class(erdf:PositivelyClosedClass).
:- defines internal normal class(erdf:NegativelyClosedClass).

:- defines internal normal class(erdf:TotalProperty).
:- defines internal normal class(erdf:PositivelyClosedProperty).
:- defines internal normal class(erdf:NegativelyClosedProperty).

:- defines internal normal property(erdf:complementOf).
```

ERDF's MWeb logic program (erdf.rb)

```
:- import( 'rdfs.rb', rulebase ).

% Equivalence of neg # and neg rdf:type (RIF compatibility)
neg ?X # ?Y :- neg ?X.[ rdf:type ->> ?Y].
neg ?X.[rdf:type ->> ?Y] :- neg ?X # ?Y.

% Inclusion of negative subclass extension
neg ?X.[rdfs:subClassOf ->> ?Y] :- neg ?X ## ?Y.

% ERDF extra rule for obtaining RDF properties
?Z.[ rdf:type ->> rdf:Property] :- neg ?_.[?Z ->> ?_] .

% ERDF / RDFS negative extension rules
neg ?Z.[ rdf:type ->> ?X] :- ?X.[ rdfs:subClassOf ->> ?Y ], neg ?Z.[rdf:type ->> ?Y].
neg ?Z1.[ ?X ->> ?Z2] :- ?X.[ rdfs:subPropertyOf ->> ?Y ], neg ?Z1.[?Y ->> ?Z2].

% ERDF specific rules
% Closed classes and properties
neg ?Z.[rdf:type ->> ?X] :-
      ( ?X.[rdf:type ->> erdf:PositivelyClosedClass] ; ?X.[rdf:type ->> erdf:TotalClass] ),
      ?Z # mw:Vocabulary, naf ?Z.[rdf:type ->> ?X].
?Z.[rdf:type ->> ?X] :-
      ( ?X.[rdf:type ->> erdf:NegativelyClosedClass] ; ?X.[rdf:type ->> erdf:TotalClass] ),
      ?Z # mw:Vocabulary, naf neg ?Z.[rdf:type ->> ?X].

neg ?Z1.[?X ->> ?Z2] :-
      ( ?X.[rdf:type ->> erdf:PositivelyClosedProperty] ; ?X.[rdf:type ->> erdf:TotalProperty] ),
      ?Z1 # mw:Vocabulary, ?Z2 # mw:Vocabulary, naf ?Z1.[?X ->> ?Z2].
?Z1.[?X ->> ?Z2] :-
   ( ?X.[rdf:type ->> erdf:NegativelyClosedProperty] ; ?X.[rdf:type ->> erdf:TotalProperty] ),
   ?Z1 # mw:Vocabulary, ?Z2 # mw:Vocabulary, naf neg ?Z1.[?X ->> ?Z2].

% Inference rule with complements
neg ?S.[ ?P ->> ?O] :- ?P.[ erdf:complementOf ->> ?Q ], ?S.[ ?Q ->> ?O].
neg ?S.[ ?P ->> ?O] :- ?Q.[ erdf:complementOf ->> ?P ], ?S.[ ?Q ->> ?O].

?S.[ ?P ->> ?O] :- ?P.[ erdf:complementOf ->> ?Q ], neg ?S.[ ?Q ->> ?O].
?S.[ ?P ->> ?O] :- ?Q.[ erdf:complementOf ->> ?P ], neg ?S.[ ?Q ->> ?O].

% ERDF axiomatic triples

erdf:complementOf.[ rdfs:domain ->> rdf:Property, rdfs:range ->> rdf:Property, rdf:type ->> rdf:Property ].

erdf:PositivelyClosedClass.[rdfs:subClassOf ->> rdfs:Class].
erdf:NegativelyClosedClass.[rdfs:subClassOf ->> rdfs:Class].
erdf:PositivelyClosedProperty.[rdfs:subClassOf ->> rdfs:Class].
erdf:NegativelyClosedProperty.[rdfs:subClassOf ->> rdfs:Class].

erdf:TotalClass.[rdfs:subClassOf ->> rdfs:Class].
erdf:TotalProperty.[rdfs:subClassOf ->> rdfs:Class].
```

Mark that declaring a class (or property) simultaneously positively and negatively closed has the same practical effect as declaring it total. The handling of closed classes and properties is a recent advancement not yet reported in the literature.

ERDF requires extra rules to take care of negative extensions for RDF and RDFS properties. Most of the rules having strong negation in the heads are in fact capturing the ERDF dual rules of RDF and RDFS entailment. Finally, by declaring two properties complementary the positive and negative instances are exchanged, as the 'erdf.rb' MWeb logic program file shows. The axiomatic triples of ERDF are also included. Blank nodes in user theories are captured by declaring them typed literals of symbol space **rif:local**, and are local to the files where they occur.

In a similar way, we expect to capture the OWL 2 RL profile by implementing the full set of OWL 2 RL/RDF rules.

6. Use

After downloading and confirming correct installation (see Section 3) it is necessary to load the system by consulting the **mweb** file. An usual session consists of compiling the rulebases, loading in the corresponding Prolog modules, and then querying the loaded rulebases. Each of these steps is now detailed.

Compilation

After developing your MWeb rulebases it is necessary to compile them. The compilation can be done independently for each rulebase and in any order. The name of the rulebase is used to generate the file names of the interface and logic program files by encoding it and appending the suffixes, '.mw' and '.rb', respectively.
A prolog module will be generated with the compiled code. The name of the Prolog file is obtained from the rulebase declaration in the interface source file. The Prolog file has .pl extension and filename obtained by escaping all chars except for letters, digits, HYPHEN-MINUS ("-") and LOW_LINE ("_") of the rulebase IRI.
A rulebase with source files 'geo.mw' and 'geo.rb' can be compiled with the command

```
mwebCompileModule( geo )
```

generating the file 'http%3A%2F%2Fgeography%2Eint.pl' if the rulebase declaration in 'geo.rw' is

```
:- rulebase 'http://geography.int'.
```

Users can also organize the rulebases by using the encoded name. The previous rulebase could be encoded in the following interface and logic program files 'http%3A%2F%2Fgeography%2Eint.mw', 'http%3A%2F%2Fgeography%2Eint.rb', respectively, and the compilation of the rulebase can be done with the command

```
mwebCompileModule( 'http://geography.int' )
```

The binary predicate `mwebCompileModule( +InterfaceFile, +RulesFile )` allows for the compilation from explicitly provided interface and logic program files. Currently, all the files should be in XSB's default search path.

Output of the compilation process is very verbose and usually some warnings might be printed. If the rulebase is not legal then an error is thrown. All these predicates compile immediately the generated XSB's Prolog file implementing MWeb's semantics.

Loading

After compiling the rulebase, each module can be loaded one at a time or by recursively loading the rulebases listed in the rulebases' uses declarations.

The single rulebase loading command is `mwebLoadModule( +RulebaseIRI )`, while the recursive one is `mwebLoadAllModules( +RulebaseIRI )`.
For instance, the command

```
mwebLoadModule( 'http://geography.int' )
```

loads the rulebase compiled in module file 'http%3A%2F%2Fgeography%2Eint.pl'. Several dynamic predicates are asserted to XSB's database in order to hide from the user the internal names used by the MWeb compilation process as well as to support properly global predicates. Users may unload explicitly modules generated from rulebases with the command `mwebUnloadModule( +RulebaseIRI )`.

Users may load several modules by using command `mwebLoadModules( +ListOfIRIs )` which loads each module appearing in the Prolog list ListOfIRIs. It can also be used to load a single module. This predicate calls `mwebLoadModule/1` for each member of the list. The command

```
mwebLoadModules( ['http://geography.int','http://europa.eu'] )
```

loads the rulebases compiled in files 'http%3A%2F%2Fgeography%2Eint.pl' and 'http%3A%2F%2Feuropa%2Eeu.pl'.

Sometimes errors while loading modules leave the MWeb's internal database in an inconsistent state. It is better that you start over again in a new instance of XSB, after fixing any existing bugs or errors.

Finally, mark that visible predicates without a rulebase list in the uses declaration, import from all the loaded modules that define such predicates. Since they have not rulebase import lists, most of the times it is necessary to explicit load the appropriate rulebases explicitly via the `mwebLoadModule/1` command.

7. Querying

The MWeb querying interface is immediate to use via the several mwebQuery predicates. For simplifying writing of queries, users may declare prefixes at the console by using the same syntax of MWeb interface files.

```
| ?- prefix eu = 'http://europa.eu#', geo = 'http://geography.int#'.

yes
```

The MWeb system declares initially useful prefixes by the following declarations:

```
:- prefix xs  = 'http://www.w3.org/2001/XMLSchema#'.
:- prefix rdf  = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
:- prefix rdfs = 'http://www.w3.org/2000/01/rdf-schema#'.
:- prefix rif = 'http://www.w3.org/2007/rif#'.
:- prefix func = 'http://www.w3.org/2007/rif-builtin-function#'.
:- prefix pred = 'http://www.w3.org/2007/rif-builtin-predicate#'.
:- prefix owl = 'http://www.w3.org/2002/07/owl#'.
```

Any prefix may be redefined by declaring again the new prefix.
The basic binary predicate `mwebQuery( +FreeVars, +Query )` executes Query in normal regime using MWebWFS semantics, and by instantiating the free variables in FreeVars with all possible combinations of constants of the several vocabularies of the loaded rulebases. The use of the free variables argument must be used when floundering occurs, i.e. calls to non ground weakly negated literals occur during the execution; in particular when **naf** is applied to non-ground literals in the query. Usually, for MWebWFS you can pass an empty list which is more efficient. If you get an execution error, then put the variables in the FreeVars list.
Naturally, the command line does not support the fancy syntax, so users should keep to Prolog conventions. Some example queries are shown below, including the RIF and ERDF syntax. The full explained examples can be found in Section 8.
We show some examples with the basic syntax using the security example rulebase:

```
% Determines the pairs of citizen X of country Y
| ?- mwebQuery( [X,Y], 'http://security.int#citizenOf'(X,Y) ).
X = Anne
Y = http://geography.int#Austria;

X = Boris
Y = http://geography.int#Croatia;
no

% Determines the countries Y which are know to not being European Union countries
| ?- mwebQuery( [], neg eu:'CountryEU'(Y) ).

Y = http://geography.int#Egypt;
Y = http://geography.int#Canada;
Y = http://geography.int#China;
Y = http://geography.int#Croatia;
```

```
% Determines the citizens X of non European Union countries.
% Notice the use of parenthesis for complex queries and @ for querying specific rulebases
| ?- mwebQuery( [], ( 'http://security.int#citizenOf'(X,Y),
                       ( neg eu:'CountryEU'(Y) ) @ 'http://europa.eu' ) ).

X = Boris
Y = http://geography.int#Croatia;

no
```

The next examples show the combination of the RIF syntax with RDF Schema support.

```
% The entities X which are countries (http://geography.int#Country)
| ?- mwebQuery( [X], X # geo : 'Country' ).

X = http://geography.int#Italy;

X = http://geography.int#Egypt;

X = http://geography.int#Croatia;

no

% The entities X which are countries (http://geography.int#Country) now using RIF frames
| ?- mwebQuery( [X], X.[rdf:type ->> geo : 'Country'] ).

X = http://geography.int#Italy;
X = http://geography.int#Egypt;
X = http://geography.int#Croatia;

no

% Finding the capitals Y of countries X
| ?- mwebQuery( [], X.[rdf:type ->> geo : 'Country', geo:capital ->> Y ] ).

X = http://geography.int#Croatia
Y = http://geography.int#Zagreb;

X = http://geography.int#Italy
Y = http://geography.int#Rome;

X = http://geography.int#Egypt
Y = http://geography.int#Cairo;

no
```

It is only possible to call rdf:type in the previous two queries because property rdf:type has been made global by one of the rulebases, otherwise no answer would be obtained. Users can also work directly with RIF frame internal predicate '->'(Attribute, Object, Value) obtaining the same results as the query Object.[Attribute ->> Value], as per the example below. If the free variables list is empty a more convenient unary version of the mwebQuery(+Query) command is available:

```
% Obtains all the ‘triples’ which have geo:'Italy' as object
| ?- mwebQuery( '->'(Pred,Subj,geo:'Italy') ).

Pred = http://www.travel.org#travel
Subj = http://www.travel_plan.gr#package1;
```

```
no

% Obtain the things X which are not RDFS classes
| ?- mwebQuery( [X], naf X # rdfs:'Class' ).

X = http://www.w3.org/2000/01/rdf-schema#comment;
X = http://www.travel.org#visit;

… (the list continues)
```

The following queries illustrate the flexibility of the system introduced by the two negations and its use.

```
% Obtain the European countries
| ?- mwebQuery( [X], ( X # geo:'Europ_Country' ) ).

X = http://geography.int#Croatia;
X = http://geography.int#Italy;

no
% Obtain the non-European countries
| ?-  mwebQuery( [X], ( X # geo:'Country', neg X # geo:'Europ_Country') ).

X = http://geography.int#Egypt;

% Obtain the entities X believed not to be European countries
| ?- mwebQuery( [X], naf X # geo:'Europ_Country' ).

X = http://www.w3.org/2000/01/rdf-schema#comment;
X = http://www.travel.org#visit;

… (the list continues)
```

More advanced uses of the querying interface may use the other two variants of the mwebQuery predicate.

The call `mwebQuery( Vars, Goal, Semantics )`, executes Goal with free variables Vars under Semantics in normal mode. Semantics is either the constant `wfs` (the default based on Well-founded Semantics - MWebWFS), or `asp` (the answer set based semantics - MWebAS). Notice that the asp version requires all variables to be grounded in the query, so all variables in the query must appear in the Vars list, and can be very inefficient since an answer set solver is called to handle the query (complexity co-NP-complete).

The most general version is `mwebQuery(+Vars,+Goal,+Semantics,+Regime)` which allows the user to use a particular MWeb regime: `d` – for definite; `o` – for open; `c` – for closed; `n` – for normal which is the default one.

All the variants of the querying predicates are further explored in the Examples section.

8. Examples

In this section we provide and explain the main features of the MWeb framework by three extended examples. One of the examples is devoted to explain the several MWeb constructs and regime modes, as well as practical differences among them (security example). The second example illustrates the support of ERDF (travel example). The remaining example illustrates inconsistency handling and global predicates in MWeb (inconsistent example).

Security Example

This example can be found in directory '`USERDIR/examples/security`' and the rulebase files can be compiled and loaded by consulting `[mweb,testSecurity]` and then calling the predicate `load`. A test can be performed by executing predicate `test`:

```
| ?- load, test.
MWebWFS solutions:
Anne

MWebAS solutions:
Boris
Anne

yes
```

The example file also declares the prefixes used in the following examples.

First queries

The first rulebase defines World countries via predicate 'http://geography.int#Country' whose instances are positively closed. In practice, this means that developers are stating that the list is complete (not true, as readers can manifestly check). Since the predicate is defined local it cannot be defined local or global in a different rulebase.

Geo's MWeb interface (geo.mw)	RIF's MWeb logic program (geo.rb)
:- rulebase 'http://geography.int'. :- base 'http://geography.int#'. :- defines local closed Country/1. :- defines global normal class(mw:Vocabulary). :- uses normal class(mw:Vocabulary).	Country(Austria). Country(Canada). Country(China). Country(Croatia). Country(Greece). Country(Egypt). Country(Portugal).

Let's check what consequences we can extract from the above rulebase, where results can be found below, assuming that :

`?- mwebQuery( geo:'Country'(X) ).` `?- mwebQuery( [X],naf neg geo:'Country'(X)).`	`?- mwebQuery( neg geo:'Country'(X) ).` `?- mwebQuery( [X], naf geo:'Country'(X) ).`
`X = http://geography.int#Portugal;` `X = http://geography.int#Greece;`	`X = Rita;` `X = Anne;`

X = http://geography.int#Canada; X = http://geography.int#China; X = http://geography.int#Egypt; X = http://geography.int#Croatia; X = http://geography.int#Austria;	X = Peter; X = Boris; X = Vocabulary;

Since the predicate is defined closed, there are no differences between objective queries and default queries because the vocabulary includes all defined constants exported by modules (because of the `uses normal class(mw:Vocabulary)` declaration). On the left, appears the list of known countries and believe to be countries (naf neg construction); on the right, appears the entities that are not countries (neg) and believed to not being countries (naf). Since the developer did not define a context predicate for geo:Country the closed world assumption is performed with respect to instances of **mw:Vocabulary** (thus, constants 'Anne', 'Boris', 'Peter' and 'Rita' appear elsewhere). Also notice the use of the base directive to simplify the encoding of the rulebase.

It is instructive to query with a new constant not in the vocabulary to observe the different behaviour:

Before adding geo:'UK' to vocabulary	After adding geo:'UK' to vocabulary
?- mwebQuery(geo:'Country'(geo:'UK')). no ?- mwebQuery(neg geo:'Country'(geo:'UK')). no ?- mwebQuery(naf geo:'Country'(geo:'UK')). yes ?- mwebQuery(naf neg geo:'Country'(geo:'UK')). yes	?- vocabulary geo:'UK'. yes ?- mwebQuery(geo:'Country'(geo:'UK')). no ?- mwebQuery(neg geo:'Country'(geo:'UK')). yes ?- mwebQuery(naf geo:'Country'(geo:'UK')). yes ?- mwebQuery(naf neg geo:'Country'(geo:'UK')). no

Since `'http://geography.int#UK'` is not listed in the rulebase it does not hold `'http://geography.int#Country'('http://geography.int#UK')`. Because the constant `'http://geography.int#UK'` does not belong to the vocabulary then closed world assumption does not apply and the query `neg 'http://geography.int#Country'('http://geography.int#UK')` fails. The weak negations of these queries succeed. We use here the fully expanded names of the prefixes in order to remind users that these are in fact the real names of the predicates and constants.
The constant `'http://geography.int#UK'` can be added to all loaded rulebases by calling vocabulary predicate, and different behaviour is obtained (see the right hand side of the previous table). By closed world assumption, it is concluded that `'http://geography.int#UK'` is not a country, and so query `neg geo:'Country'(geo:'UK')` succeeds and `naf neg geo:'Country'(geo:'UK')` fails.

Vocabulary can also be added to a particular rulebase. E.g., a call of the form `vocabulary geo:'UK' @ 'http://geography.int'` would have the same effect in the previous example.

The different reasoning modes

The 'http://geography.int' rulebase can also be queried in other reasoning modes, illustrating the differences among them. The definite regime is very similar to ordinary Prolog, as users can check below:

Objective queries on regime d	Default queries on regime d
`|?- mwebQuery( [X],` `geo:'Country'(X), wfs, d ).` `X = http://geography.int#Portugal;` `X = http://geography.int#Greece;` `X = http://geography.int#Canada;` `X = http://geography.int#China;` `X = http://geography.int#Egypt;` `X = http://geography.int#Croatia;` `X = http://geography.int#Austria;` `no` `| ?- mwebQuery( [X],` `neg geo:'Country'(X), wfs, d ).` `no` `| ?- mwebQuery( [],` `geo:'Country'('France'), wfs, d ).` `no` `| ?- mwebQuery( [], neg` `geo:'Country'('France'), wfs, d ).` `no`	`| ?- mwebQuery( [X],` `naf geo:'Country'(X), wfs, d ).` `X = Rita;` `X = Anne;` `X = Peter;` `X = Boris;` `X = http://geography.int#UK;` `X = Vocabulary;` `no` `| ?- mwebQuery( [X],` `naf neg geo:'Country'(X), wfs, d ).` `X = Rita;` `X = http://geography.int#Portugal;` `X = http://geography.int#UK;` `X = Anne;` `X = http://geography.int#Greece;` `X = http://geography.int#Canada;` `X = Peter;` `X = http://geography.int#China;` `X = Boris;` `X = http://geography.int#Egypt;` `X = http://geography.int#Croatia;` `X = http://geography.int#Austria;` `X = Vocabulary;` `no` `| ?- mwebQuery( [],` `naf geo:'Country'('France'), wfs, o ).` `yes` `| ?- mwebQuery( [],` `naf neg geo:'Country'('France'), wfs, o ).` `yes`

As expected, we obtain the list of answers of registered countries. Notice that the default query `naf geo:'Country'(X)` obtains the complementary solutions of the registered vocabulary. Since no rules for `neg geo:'Country'(X)` are present in the logic program the query fails with no answers, and thus all the constants in the vocabulary are answers to the default query `naf neg geo:'Country'(X)`. It can be shown that objective answers are monotonic on the extension of the rulebase, while default ones are non-monotonic. This

justifies the restriction to forbid the use of weak negation in the bodies of rules. A query using a constant (`'France'`) not in the vocabulary has identical behaviour.

The remaining distinctive mode to analyse is the open entailment regime, explained next.

Objective queries on regime o	Default queries on regime o
`| ?- mwebQuery( [X],` `  geo:'Country'(X), wfs, o ).` `X = http://geography.int#Portugal;` `X = http://geography.int#Greece;` `X = http://geography.int#Canada;` `X = http://geography.int#China;` `X = http://geography.int#Egypt;` `X = http://geography.int#Croatia;` `X = http://geography.int#Austria;` `no` `| ?- mwebQuery( [X],` `  neg geo:'Country'(X), wfs, o ).` `no` `| ?- mwebQuery( [],` `  geo:'Country'('France'), wfs, o ).` `no` `| ?- mwebQuery( [],` `  neg geo:'Country'('France'), wfs, o ).` `no`	`| ?- mwebQuery( [X],` `  naf geo:'Country'(X), wfs, o ).` `no` `| ?- mwebQuery( [X],` `  naf neg geo:'Country'(X), wfs, o ).` `X = http://geography.int#Portugal;` `X = http://geography.int#Greece;` `X = http://geography.int#Canada;` `X = http://geography.int#China;` `X = http://geography.int#Egypt;` `X = http://geography.int#Croatia;` `X = http://geography.int#Austria;` `no` `| ?- mwebQuery( [],` `  naf geo:'Country'('France'), wfs, o ).` `yes` `| ?- mwebQuery( [],` `  naf neg geo:'Country'('France'), wfs, o ).` `yes`

Regarding objective queries the results are as before. The default queries are very interesting to explain. Roughly, all the things that are not entailed by the objective part are undefined (note the symmetries). This mode is monotonic for objective queries, and thus is the appropriate mode for safe reasoning in the Semantic Web. Moreover, under MWebAS semantics it can generate alternative worlds and do reasoning by cases. However, it is less efficient than definite mode for both MWebWFS and MWebAS semantics.
Since constant `'France'` does not belong to the vocabulary (context predicate fails) then the same behaviour of the other modes is obtained. Therefore, undeclared vocabulary can bring semantic problems: do not forget to declare all the vocabulary!

To conclude discussion, if the predicate `geo:'Country'/1` is queried in normal mode then the used mode is in fact closed according to Table 4 because the predicate has been defined in closed mode. Do not forget that this table is always used to obtain the real mode used to perform the query.

The different semantics

Consider now the rulebase defining the European countries, built on top of the geography rulebase. The list of world countries is used to provide the context predicate for performing the closed world assumptions; in all respects this is very similar to the geography rulebase.

Europa's MWeb interface (europa.mw)	Europa's MWeb logic program (europa.rb)
:- rulebase 'http://europa.eu'. :- prefix eu = 'http://europa.eu#', geo = 'http://geography.int#'. :- defines local closed eu:CountryEU/1 wrt context geo:Country/1. :- uses definite geo:Country/1 from 'http://geography.int'. :- uses normal class(mw:Vocabulary) from 'http://geography.int'.	eu : CountryEU(geo:Austria). eu : CountryEU(geo:Greece). eu : CountryEU(geo:Portugal).

Some interesting queries are collected below (the order of the answers does not matter):

Objective queries	Default queries
`` | ?- mwebQuery('http://europa.eu#CountryEU'(C)). C = http://geography.int#Austria; C = http://geography.int#Greece; C = http://geography.int#Portugal; no | ?- mwebQuery(neg 'http://europa.eu#CountryEU'(C)). C = http://geography.int#Canada; C = http://geography.int#China; C = http://geography.int#Croatia; C = http://geography.int#Egypt; ```	``` | ?- mwebQuery([C], naf 'http://europa.eu#CountryEU'(C)). C = Rita; C = http://geography.int#UK; C = Anne; C = http://geography.int#Canada; C = Peter; C = http://geography.int#China; C = Boris; C = http://geography.int#Egypt; C = http://geography.int#Croatia; C = Vocabulary; no | ?- mwebQuery([C], naf neg 'http://europa.eu#CountryEU'(C)). C = Rita; C = http://geography.int#Portugal; C = http://geography.int#UK; C = Anne; C = http://geography.int#Greece; C = Peter; C = Boris; C = http://geography.int#Austria; C = Vocabulary; no ```

Since an explicit context predicate was provided, then the closed world assumptions are done with respect to the answers of this predicate, thus limiting the things that are known to not being European countries. With negation as failure more answers are obtained, but are non-monotonic (later on we might discover that UK is in fact an European Country).

The rulebase 'http://security.int' defines citizenship and suspects of criminal acts. Both predicates are defined open. However, citizenship relations are only visible to the rulebase identified by 'http://gov.country'.

Security's MWeb interface (security.mw)	Security's MWeb logic program (security.rb)
:- rulebase 'http://security.int'. :- prefix sec = 'http://security.int#', geo = 'http://geography.int#'. :- defines local open sec:citizenOf/2 visible to 'http://gov.country'. :- defines global open sec:Suspect/1. :- defines global normal class(mw:Vocabulary). :- uses normal class (mw:Vocabulary).	sec:citizenOf(Anne, geo:Austria). sec:citizenOf(Boris, geo:Croatia). sec:Suspect(Peter). neg sec:Suspect(Rita).

The relevant queries to our example are found below, and the expected behaviour is obtained since all predicates are declared open.

Objective queries	Default queries
| ?- mwebQuery([P,C], sec:citizenOf(P,C)). P = Anne C = http://geography.int#Austria; P = Boris C = http://geography.int#Croatia; no | ?- mwebQuery([P,C], neg sec:citizenOf(P,C)). no | ?- mwebQuery([P], sec:'Suspect'(P)). P = Peter; no | ?- mwebQuery([P], neg sec:'Suspect'(P)). P = Rita; no	| ?- mwebQuery([P,C], naf sec:citizenOf(P,C)). no | ?- mwebQuery([P,C], naf neg sec:citizenOf(P,C)). P = Anne C = http://geography.int#Austria; P = Boris C = http://geography.int#Croatia; no | ?- mwebQuery([P], naf sec:'Suspect'(P)). P = Rita; no | ?- mwebQuery([P], naf neg sec:'Suspect'(P)). P = Peter; no

Finally, rulebase 'http://gov.country' expresses some rules to allow entering into an imaginary country resorting to all the previous rulebases.

Gov's MWeb interface (gov.mw)	Gov's MWeb logic program (gov.rb)
:- rulebase 'http://gov.country'. :- prefix gov = 'http://gov.country#'. :- prefix eu = 'http://europa.eu#'. :- prefix geo = 'http://geography.int#'. :- prefix sec = 'http://security.int#'. :- defines local normal gov:Enter/1 visible to 'http://security.int'. :- defines local closed neg gov:RequiresVisa/1 wrt context geo:Country/1. :- uses definite geo:Country/1 from 'http://geography.int'. :- uses open eu:CountryEU/1 from 'http://europa.eu'. :- uses definite sec:citizenOf/2, sec:Suspect/1 from 'http://security.int'. :- uses normal class(mw:Vocabulary).	:- defines internal open sec:citizenOf/2. gov:Enter(?P) :- eu:CountryEU(?C), sec:citizenOf(?P,?C), ~ sec:Suspect(?P) @ 'http://security.int'. gov:Enter(?P) :- - eu:CountryEU(?C), sec:citizenOf(?P,?C), - gov:RequiresVisa(?C), ~ sec:Suspect(?P) @ 'http://security.int'. -gov:RequiresVisa(geo:Croatia). sec:citizenOf(Peter,geo:Greece).

The rulebase shows other important features of the MWeb framework. First, users can always refine existing predicates by using it in definite mode and defining it internal on the wished regime; extra facts and rules can be defined internally in the rulebase (see `sec:citizenOf/2` predicate). A predicate defined in normal mode can have occurrences of weak negation in the bodies of its rules, as the `gov:Enter/1` predicate shows. A rulebase can query a specific rulebase by using the '@' operator.

This last rulebase also helps to understand the differences between MWebWFS and MWebAS semantics. Let's check which persons are allowed to enter the country:

Answers under MWebWFS semantics	Answers under MWebAS semantics
| ?- mwebQuery([P], 'http://gov.country#Enter'(P), wfs). P = Anne; no | ?- mwebQuery([P], neg 'http://gov.country#Enter'(P)). no | ?- mwebQuery([P], naf 'http://gov.country#Enter'(P)). P = Peter; no | ?- mwebQuery([P], naf neg 'http://gov.country#Enter'(P)). P = Rita; P = http://geography.int#Portugal; P = http://geography.int#UK; P = Anne; P = http://geography.int#Greece; P = http://geography.int#Canada; P = Peter; P = http://geography.int#China; P = Boris; P = http://geography.int#Egypt; P = http://geography.int#Croatia; P = http://geography.int#Austria; P = Vocabulary; no	| ?- mwebQuery([P], 'http://gov.country#Enter'(P), asp). P = Anne; P = Boris; no | ?- mwebQuery([P], neg 'http://gov.country#Enter'(P), asp). no | ?- mwebQuery([P], naf 'http://gov.country#Enter'(P), asp). P = Peter; no | ?- mwebQuery([P], naf neg 'http://gov.country#Enter'(P), asp). P = Rita; P = http://geography.int#Portugal; P = http://geography.int#UK; P = Anne; P = http://geography.int#Greece; P = http://geography.int#Canada; P = Peter; P = http://geography.int#China; P = Boris; P = http://geography.int#Egypt; P = http://geography.int#Croatia; P = http://geography.int#Austria; P = Vocabulary; no

Under MWebWFS and MWebAS we conclude that Anne is allowed to enter the country, since she is a citizen of Austria, an European country, and she is not believed suspect of any criminal actions. However, we can only conclude that Boris is allowed to enter the country using MWebAS. This is so because we need to do some case analysis (consider multiple possible models).
It is known that Boris is a citizen of Croatia and is not suspect. However we do not know if Croatia is a European country. Since predicate `eu:CountryEU/1` is used in open mode, the MWebAS semantics considers a model where Croatia is an European country, and thus Boris can enter by the first rule, and considers another model where Croatia is not an European country, but since it is not required a Visa for Croatia, Boris can also enter the country in this case by the second rule. Since, in all possibilities Boris can enter the country by cautious reasoning we conclude that it is safe to allow Boris enter the country.
Since there are no rules for `neg gov:Enter/1` and predicate is defined in normal mode, there is no way to conclude explicitly false knowledge. However, we are able to conclude that the system believes that Peter should not enter the country (because he is a suspect).

Regarding Rita we cannot conclude anything since she might be a citizen of an European country or not. Therefore, she might be allowed to enter in some models, and refused to enter in other models (where she has non-European nationality).

Since reasoning in MWebWFS considers only one model, reasoning is polynomial on the size of the grounded rulebase. In the worst case, reasoning in MWebAS is co-NP-complete (intractable) since it might be necessary to consider an exponential number of possible models.

Travel Example

The travel example can be found in directory '`USERDIR/examples/travel`' and the rulebase files can be compiled and loaded by consulting `[mweb,testTravel]` and then calling the predicate `load`. Since this example can consume quite a large amount of memory with MWebAS semantics, users are advised to invoke XSB with parameter –m 50000. A simple test can be performed by executing predicate `test`:

```
| ?- load, test.
MWebWFS solutions:
(http://www.travel_plan.gr#package2  ','  http://geography.int#Croatia)
(http://www.pyramis.gr#package1  ','  http://geography.int#Egypt)

MWebAS solutions:
(http://www.travel_plan.gr#package2  ','  http://geography.int#Croatia)
(http://www.pyramis.gr#package1  ','  http://geography.int#Egypt)

yes
```

A more extensive test can be executed by calling predicate testall, dumping the list of triples (Subject,Property,Object) which can be concluded from the example under MWebWFS and MWebAS semantics. Users can evidence the different response times of both semantics (MWebWFS is faster than MWebAS).

Using ERDF syntax and semantics

We start again by defining a rulebase with world and European countries. This time we use the ERDF support of the MWeb framework, which introduces new syntax for simplifying writing of rulebases on top of the (Extended) Resource Description Framework.

Geo's RDF MWeb interface (geordf.mw)	Geo's RDF MWeb logic program (geordf.rb)
:- rulebase 'http://geography.int'. :- base 'http://geography.int'. :- prefix geo = 'http://geography.int#'. :- import('erdf.mw',interface). :- defines local normal class(geo:Country). :- defines local normal class(geo:Europ_Country). :- defines local normal property(geo:capital).	:- import('erdf.rb',rulebase). geo:Europ_Country.[rdf:type ->> erdf:PositivelyClosedClass]. geo:Europ_Country.[rdfs:subClassOf ->> '#Country']. geo:Egypt.[rdf:type ->> geo:Country]. geo:Egypt.[geo:capital ->> geo:Cairo]. geo:Italy.[rdf:type ->> geo:Europ_Country, geo:capital ->> geo:Rome]. geo:Croatia.[rdf:type ->> geo:Europ_Country, geo:capital ->> geo:Zagreb].

The rulebase defines properties and classes. In fact, only classes that require context literals must be declared. However, properties for which we have rules must always be declared. A class declaration `class(CURIE)` is short for `?_ # mw:Vocabulary` (coinciding with`?_.[rdf:type ->> CURIE]`). Recall that in MWeb ERDF all classes and properties normal, and notice that the rdf(s) properties are declared elsewhere and imported both in the interface file (declarations) and logic program (rules). Observe also how triples (frames) for the same subject can be aggregated together. The intended semantics of this rulebase is also pretty clear: we have a positively closed list of European Countries, some instances are listed as well their capitals.

Some interesting queries illustrating some features are shown next:

ERDF queries using RIF frames	ERDF queries using RIF relations
| ?- mwebQuery([X], X.[rdf:type ->> geo:'Country']). X = http://geography.int#Italy; X = http://geography.int#Croatia; X = http://geography.int#Egypt; no | ?- mwebQuery([X], neg X.[rdf:type ->> geo:'Country']). no | ?- mwebQuery([X], X.[rdf:type ->> geo:'Europ_Country']). X = http://geography.int#Italy; X = http://geography.int#Croatia; no | ?- mwebQuery([X], (X.[rdf:type ->> geo:'Country'], neg X.[rdf:type ->> geo:'Europ_Country'])). X = http://geography.int#Egypt; no | ?- mwebQuery([X,Y], X.[rdf:type ->> geo:'Europ_Country', geo:capital ->> Y]). X = http://geography.int#Italy Y = http://geography.int#Rome; X = http://geography.int#Croatia Y = http://geography.int#Zagreb; no | ?- mwebQuery([X], geo:'Europ_Country'.[rdfs:subClassOf ->> X]). no;	| ?- mwebQuery([X], X # geo:'Country'). X = http://geography.int#Italy; X = http://geography.int#Croatia; X = http://geography.int#Egypt; no | ?- mwebQuery([X], neg X # geo:'Country'). no | ?- mwebQuery([X], X # geo:'Europ_Country'). X = http://geography.int#Italy; X = http://geography.int#Croatia; no | ?- mwebQuery([X], (X # geo:'Country', neg X # geo:'Europ_Country')). X = http://geography.int#Egypt; no | ?- mwebQuery([X,Y], (X # geo:'Europ_Country', X.[geo:capital ->> Y])). X = http://geography.int#Italy Y = http://geography.int#Rome; X = http://geography.int#Croatia Y = http://geography.int#Zagreb; no | ?- mwebQuery([X], geo:'Europ_Country' ## X). no

The expected answers are obtained. However the last query deserves some explanation. Using `rdfs:subClassOf` one does not obtain answers because the property is declared internal and thus is not visible. No naswer is obtained using `'##'/2` because `'##'/2` is included in `rdfs:subClassOf` but not vice-versa and the predicate is also invisible.
Be very careful when negating RIF frame formulas, since the current parser only supports negation of frames with a single property. So, users cannot query `naf X.[rfds:subClassOf ->> Y, Y rdfs:subClassOf Z]`.

Total and closed classes/properties

The rulebase 'http://europa.eu' defines an open (and incomplete) list of European Union countries. The open world assumption is declared by the statement that 'http://europa.eu#CountryEU' is an erdf:TotalClass. A class closed world assumption can be declared in a similar manner, by erdf:PositivelyClosedClass and erdf:NegativelyClosedClass, as shown in the previous geordf.rb MWeb logic program.

Europa's MWeb interface (europardf.mw)	Europa's MWeb logic program (europardf.rb)
:- rulebase 'http://europa.eu'. :- base 'http://europa.eu#'. :- prefix eu = 'http://europa.eu#', geo = 'http://geography.int#'. :- import('erdf.mw',interface). :- defines local normal class(eu:CountryEU). :- uses normal class(geo:Country) from 'http://geography.int'. :- uses normal class(mw:Vocabulary) from 'http://geography.int'.	:- import('erdf.rb',rulebase). eu:CountryEU.[rdf:type ->> erdf:TotalClass]. geo:Italy.[rdf:type ->> eu:CountryEU]. geo:Greece.[rdf:type ->> eu:CountryEU].

To check that the European Union class is really open, users can perform the following queries to contrast the different behaviour of European countries according to 'http://europa.eu' rulebase and 'http://geography.int' .

Checking open class	Checking closed class
| ?- mwebQuery([X], X # eu:'CountryEU'). X = http://geography.int#Italy; X = http://geography.int#Greece; no | ?- mwebQuery([X], neg X # eu:'CountryEU'). no | ?- mwebQuery([X], naf X # eu:'CountryEU'). X = http://www.travel.org#visit; X = http://www.pyramis.gr#package2; X = http://www.travel_plan.gr#package2; X = http://geography.int#Luxor; X = http://www.travel.org#travel; X = http://www.pyramis.gr#package1; X = http://www.travel_plan.gr#package1; X = http://www.anne_travel_pref.gr#choose_trav_package; X = http://geography.int#Trogir; X = http://www.anne_travel_pref.gr#visit_other; no | ?- mwebQuery([X], naf neg X # eu:'CountryEU'). X = http://geography.int#Italy; X = http://geography.int#Greece; (plus the solutions of the previous query) no	| ?- mwebQuery([X], X # geo:'Europ_Country'). X = http://geography.int#Italy; X = http://geography.int#Croatia; no | ?- mwebQuery([X], neg X # geo:'Europ_Country'). X = http://erdf.org#complementOf; X = http://www.w3.org/2000/01/rdf-schema#domain; X = http://www.w3.org/2000/01/rdf-schema#comment; X = http://www.w3.org/1999/02/22-rdf-syntax-ns#_1; X = http://geography.int#Country; … no | ?- mwebQuery([X], naf X # geo:'Europ_Country'). X = http://erdf.org#complementOf; X = http://www.w3.org/2000/01/rdf-schema#domain; X = http://www.travel.org#visit; X = http://www.w3.org/2000/01/rdf-schema#comment; X = http://www.w3.org/1999/02/22-rdf-syntax-ns#_1; … | ?- mwebQuery([X], naf neg X # geo:'Europ_Country'). X = http://www.travel.org#visit; X = http://geography.int#Greece; X = http://www.pyramis.gr#package2; …

Since 'http://geography.int#Europ_Country' is closed with respect to the vocabulary of 'http://geography.int', the solutions of the last two queries in open mode present some solutions with vocabulary of other rulebases. ERDF always uses the whole available vocabulary for open and closed world assumptions.

The declaration of total properties is done in the same way, by stating that the property has rdf:type erdf:TotalProperty. Closed properties are declared using erdf:PositivelyClosedProperty and erdf:NegativelyClosedProperty, depending on the intended semantics.

Global predicates

Travel packages are expressed in the following two rulebases 'http://www.pyramis.gr' and 'http://www.travel_plan.gr', whose definition can be found next.

Pyramis rulebase	Travel rulebase
:- rulebase 'http://www.pyramis.gr'. :- base 'http://www.pyramis.gr'. :- prefix geo = 'http://geography.int#', vac = 'http://www.travel.org#', pyr = 'http://www.pyramis.gr#'. :- import('erdf.mw',interface). :- defines global normal property(vac:travel). :- defines global normal property(vac:visit).	:- rulebase 'http://www.travel_plan.gr'. :- base 'http://www.travel_plan.gr'. :- prefix geo = 'http://geography.int#', vac = 'http://www.travel.org#', trav = 'http://www.travel_plan.gr#'. :- import('erdf.mw',interface). :- defines global normal property(vac:travel). :- defines global normal property(vac:visit).
:- import('erdf.rb',rulebase). pyr:package1.[vac:travel ->> geo:Egypt , vac:visit ->> geo:Cairo]. pyr:package2.[vac:travel ->> geo:Egypt, vac:visit ->> geo:Cairo, vac:visit ->> geo:Luxor].	:- import('erdf.rb',rulebase). trav:package1.[vac:travel ->> geo:Italy , vac:visit ->> geo:Rome]. trav:package2.[vac:travel ->> geo:Croatia, vac:visit ->> geo:Zagreb, vac:visit ->> geo:Trogir].

We can query to obtain which packages travel to which countries independently where they are defined:

```
| ?- mwebQuery( [X,Y], X.[ vac:travel ->> Y ] ).

X = http://www.pyramis.gr#package1
Y = http://geography.int#Egypt;

X = http://www.pyramis.gr#package2
Y = http://geography.int#Egypt;

X = http://www.travel_plan.gr#package2
Y = http://geography.int#Croatia;

X = http://www.travel_plan.gr#package1
Y = http://geography.int#Italy;
```

Notice that the solutions obtained by global predicates are simply the union of the answers of each loaded rulebase where they are defined. Thus, interpretations of predicates may be distinct in distinct rulebases. If a global interpretation is desired for some rulebase, that rulebase must use the predicate besides defining it. This fine-grained control is very flexible and general.

Coding complex rules

Rulebase 'http://www.anne_travel_pref.gr' expresses Anne's travel preferences. Anne's rulebase defines only one predicate, in normal mode, which is visible only to Peter's rulebase. The logic program captures the following conditions to select travel packages:

1. A package is chosen by Anne if it travels to a non-European country (according to geography's rulebase) and visits only one city according to pyramis travel agency. This is captured by the first and the second rules in the program;
2. A trip to an EU Country by Travel Plan which visits at least two cities (third rule);
3. A trip to an European but not a EU Country that visits the capital of the country (fourth rule).

Anne's MWeb interface (anne.mw)	Anne's MWeb logic program (anne.rb)
:- rulebase 'http://www.anne_travel_pref.gr'. :- base 'http://www.anne_travel_pref.gr'. :- prefix ann = 'http://www.anne_travel_pref.gr#'. :- prefix geo = 'http://geography.int#', eu = 'http://europa.eu#'. :- prefix vac='http://www.travel.org#', trav='http://www.travel_plan.gr#'. :- import('erdf.mw',interface). :- defines local normal property(ann:choose_trav_package) visible to 'http://www.peter_travel_pref.gr'. :- uses normal property(vac:travel). :- uses normal property(vac:visit). :- uses normal property(geo:capital) from 'http://geography.int'. :- uses normal class(geo:Europ_Country) from 'http://geography.int'. :- uses normal class(eu:CountryEU) from 'http://europa.eu'. :- uses normal class(mw:Vocabulary).	:- import('erdf.rb',rulebase). ?Package.[ann:choose_trav_package ->> ?Country] :- neg (?Country # geo:Europ_Country), ?Package.[vac:travel ->> ?Country], ?Package.[vac:visit ->> ?City] @ 'http://www.pyramis.gr', naf neg ?Package.[ann:visit_other ->> ?City]. neg ?Package.[ann:visit_other ->> ?City] :- ?Package.[vac:visit ->> ?City] @ 'http://www.pyramis.gr', ?Package.[vac:visit ->> ?Other] @ 'http://www.pyramis.gr', naf(?City = ?Other). ?Package.[ann:choose_trav_package ->> ?Country] :- ?Package.[vac:travel ->> ?Country], ?Package.[vac:visit ->> ?City1] @ 'http://www.travel_plan.gr', ?Package.[vac:visit ->> ?City2] @ 'http://www.travel_plan.gr', naf (?City1 = ?City2). ?Package.[ann:choose_trav_package ->> ?Country] :- ?Country.[rdf:type ->> geo:Europ_Country], neg ?Country.[rdf:type ->> eu:CountryEU], ?Package.[vac:travel ->> ?Country], ?Package.[vac:visit ->> ?City], ?City.[geo:capital ->> ?Country].

Since all properties and classes are normal, then negation as failure can be used to capture universal quantification. This allows for greater generality but with no monotonicity guarantees.

Mark that the interface uses normal properties vac:travel and vac:visit without stating their provenance, while in the logic program part of the rulebase some of the queries mention explicitly the queried rulebase. In the latter case, the corresponding rulebases must be loaded.

The selected packages by Anne can be obtained by calling predicate **test** of testTravel. If users wish to see all the entailed triples then they can enter the query **testall**.
Let's obtain the selected packages:

```
| ?- mwebQuery( [Package,Country], Package.[ann:choose_trav_package ->> Country ] ).

Package = http://www.pyramis.gr#package1
Country = http://geography.int#Egypt;

Package = http://www.travel_plan.gr#package2
Country = http://geography.int#Croatia;

no
```

The first solution is obtained by the first condition (captured by the first and second rules), and the second solution is obtained from the second condition (captured by the third rule). The first package of Travel Plan is not selected because Italy is a European Union country, and thus condition neg 'http://geography.int#Italy'.[rdf:type ->> eu:CountryEU] fails.

Inconsistent example

This example can be found in directory 'USERDIR/examples/inconsistent and the rulebase files can be compiled by consulting [mweb,testInconsistent], and loaded with the command load. A test can be performed by executing predicate test:

```
| ?- load, test.
MWebWFS solutions
(neg 'http://example.org#q'(?X)) @ 'http://example3.org'
?X = b
?X = c
'http://example.org#p'(?X)
?X = a
?X = c
naf 'http://example.org#p'(?X)
?X = b
?X = c
('http://example.org#p'(?X), 'http://example.org#q'(?X))
?X = a
?X = c
'http://example.org#r'(?X)
?X = c

MWebAS solutions:
(neg 'http://example.org#q'(?X)) @ 'http://example3.org'
?X = b
?X = c
'http://example.org#p'(?X)
++Error[XSB/Runtime] Unhandled Exception: mweb(Inconsistent modular rulebases!)
```

This example is formed by three rulebases depicted below:

Rulebase s1	**Rulebase s2**	**Rulebase s3**
:- rulebase 'http://example1.org'. :- prefix ex='http://example.org#'. :- defines local normal ex:p/1. :- defines global normal ex:q/1. :- uses normal ex:q/1.	:- rulebase 'http://example2.org'. :- prefix ex='http://example.org#'. :- defines global definite ex:q/1. :- defines local normal ex:r/1.	:- rulebase 'http://example3.org'. :- prefix ex='http://example.org#'. :- defines global normal ex:q/1.
ex:p(?X) :- ex:q(?X).	ex:q(c). neg ex:r(c). ex:r(c).	ex:q(a). neg ex:q(b). neg ex:q(c).

The definition of global predicates in normal (or closed) modes is forbidden by the theoretical MWeb framework due to non-monotonicity. For compatibility with RIF and the underlying Prolog framework, these are allowed in the implementation. Users are warned in the console when such potentially problematic situations are found.

Detecting inconsistencies

By design the implementation of the MWebWFS semantics does not check consistencies automatically, because it uses a paraconsistent semantics, while MWebAS performs checks before executing goals but taking into account dependencies of the queried goals. For

checking consistency under MWebWFS semantics on normal regime, users can call `mwebCheckConsistency`:

```
| ?- mwebCheckConsistency.
++Error[XSB/Runtime] Unhandled Exception: mweb(Inconsistent modular rulebases!)
```

It is also available the predicate `mwebCheckConsistency( +Semantics, +Regime)` for checking general consistency under Semantics in mode Regime.

```
 | ?- mwebCheckConsistency( wfs, d ).

yes
| ?- mwebCheckConsistency( asp, d ).

yes
| ?- mwebCheckConsistency( asp, n ).
++Error[XSB/Runtime] Unhandled Exception: mweb(Inconsistent modular rulebases!)
```

The results are expected since inconsistencies arise only on normal mode because of the rules for `ex:r/1` and `ex:q/1`. Notice that in definite mode only ex:q(c) is concluded under both semantics:

```
| ?- mwebQuery( [X], ex:q(X) @ 'http://example2.org', wfs, d ).

X = c;

no
| ?- mwebQuery( [X], ex:q(X) @ 'http://example2.org', asp, d ).

X = c;

no
```

One of the interesting features of MWebWFS semantics is that it is able to identify which conclusions depend on contradiction. Users can always check if some goal depends on contradiction by querying the goal and its weak negation. If both succeed then the conclusion is dependent on contradiction. This is illustrated quite well with the following queries:

```
| ?- mwebQuery( [X], ex:p(X) ).
X = a;
X = c;

no
| ?- mwebQuery( [X], neg ex:p(X) ).

no
| ?- mwebQuery( [X], naf ex:p(X) ).
X = b;
X = c;
X = Vocabulary;
no

| ?- mwebQuery( [X], naf neg ex:p(X) ).
X = a;
X = b;
X = c;
X = Vocabulary;
no
```

From the above results it can be concluded that conclusion `'http://example.org#p'(c)` depends on contradiction, since both `'http://example.org#p'(c)` and `naf 'http://example.org#p'(c)` hold but is not itself contradictory because the conclusion `neg 'http://example.org#p'(c)` does not hold.

Let's compare to what happens to predicate `ex:q/1`:

```
| ?- mwebQuery( [X], ex:q(X) ).
X = a;
X = c;

no
| ?- mwebQuery( [X], neg ex:q(X) ).
X = b;
X = c;

no
| ?- mwebQuery( [X], naf ex:q(X) ).
X = b;
X = c;

no
| ?- mwebQuery( [X], naf neg ex:q(X) ).
X = a;
X = c;

no
```

Notice that `'http://example.org#p'(c)` and `neg 'http://example.org#p'(c)` hold, thus we have contradiction. This can be checked directly or by decting the dependencies on contradiction since by coherence `naf 'http://example.org#p'(c)` and `naf neg 'http://example.org#p'(c)` both hold, as expected.

As shown before, MWebWFS is tolerant to the existence of contradictions but is capable of signalling their effects. MWebAS is not paraconsistent but in some circumstances is capable of isolating independent parts of the rulebases avoiding trivialization in expected situations.

Global and local models

The MWeb framework is designed in such a way that each rulebase has its own interpretation of all defined and used predicates, for each regime. This prevents unintended interactions of loaded modules and trivialization because of inconsistent rulebases, as illustrated previously.

For simplifying the construction of full models of rulebases the predicate mwebModel(Rulebase, Literal) and its variants can be used to backtrack over all Literals true in Rulebase. These can be collected with setof or findall:

```
| ?- setof( L, mwebModel( 'http://example3.org', L ), Model ).

L = _h140
[http://example.org#q(a),naf http://example.org#q(Vocabulary),naf
http://example.org#q(b),naf http://example.org#q(c),naf neg
http://example.org#q(Vocabulary),naf neg http://example.org#q(a),neg
http://example.org#q(b),neg http://example.org#q(c)];
```

no

The previous command obtains all the literals L true in rulebase 'http://example3.org' under semantics MWebWFS on normal mode, and collects them in list Model. The models of the remaining rulebases can be obtained similarly, and are shown in the next table, ignoring the Vocabulary constant:

Rulebase 'http://example1.org'	**Rulebase 'http://example2.org'**	**Rulebase 'http://example3.org'**
http://example.org#p(a), http://example.org#p(c), http://example.org#q(a), http://example.org#q(c), neg http://example.org#q(b), neg http://example.org#q(c), naf http://example.org#p(b), naf http://example.org#p(c), naf http://example.org#q(b), naf http://example.org#q(c), naf neg http://example.org#p(a), naf neg http://example.org#p(b), naf neg http://example.org#p(c), naf neg http://example.org#q(a), naf neg http://example.org#q(c)	http://example.org#q(c), http://example.org#r(c), neg http://example.org#r(c), naf http://example.org#q(a), naf http://example.org#q(b), naf http://example.org#r(a), naf http://example.org#r(b), naf http://example.org#r(c), naf neg http://example.org#q(a), naf neg http://example.org#q(b), naf neg http://example.org#q(c), naf neg http://example.org#r(a), naf neg http://example.org#r(b), naf neg http://example.org#r(c),	http://example.org#q(a), neg http://example.org#q(b), neg http://example.org#q(c), naf http://example.org#q(b), naf http://example.org#q(c), naf neg http://example.org#q(a)

These models clearly show that rulebase 'http://example3.org' is consistent, rulebase 'http://example2.org' is inconsistent because of predicate r defined in normal model (but consistent in definite mode since q is defined in this rulebase definite), and rulebase 'http://example1.org' is inconsistent because it uses the global predicate q which is inconsistent because of conflicting definitions between the other two rulebases. As explained before, predicate p in the first rulebase depends on q and thus has an instance inconsistent.

This example also shows how users can share the global definitions of predicates: they must use the predicate from everywhere, besides declaring it global. The global model of the loaded rulebases can also be obtained by the mwebModel(+Literal) predicate. However, results must be interpreted carefully since, as in the present example, not all rulebases may have the same definition for global predicates. For generating the same interpretation for global predicates in all rulebases, these must be defined with the same mode and with the same context, and used with the same mode in all rulebases.

9. Current limitations and future developments

For ease of implementation or more flexibility there are some differences to the theoretical MWeb framework which are collected next:

- Users can define global predicates in non-monotonic modes closed and normal. This is forbidden in the MWeb theoretical framework since these are expected to be monotonic. Users should use this at their own risk. A warning message is printed.
- The vocabulary of any rulebase is always the one obtained from all the loaded rulebases. The MWeb ERDF theoretical framework obtains the vocabulary from the dependencies of a given rulebase. For ease of implementation this has not been implemented, but it is expected to be tackled in the near future.
- Goals using vocabulary not defined in the loaded rulebases usually are false by default instead of raising any kind of runtime error. Forms to circumvent this problem have been discussed in the examples.
- Consistency checks are not performed automatically in MWebWFS. The users must perform the checks explicitly. A paraconsistent semantics is implemented with a better behaviour in face of contradiction than the MWebWFS theoretical framework.

The current implementation has several limitations, which are briefly described:

- There is no means for using the local rulebase vocabulary as context for open and closed world assumptions. More complex vocabulary constant using complex terms is necessary (currently is limited to constants).
- Subsumptive tabling is much better in terms of memory consumption than variant tabling, in particular for RDF reasoning. However, the current implementation is not handling it appropriately because of some cuts in the generated code. Alternative coding must be devised.
- There is no tabling control by the user, therefore in some situations we are using more memory than the one strictly necessary. This has negative performance impacts.
- A unique general program transformation is used both for MWebWFS and MWebAS semantics. For the case of consistent theories, the transformed program can be simplified reducing tabling.
- Elementary support of RIF and XML Schema datatypes just recognizing terms of the form Literal^^IRI.
- The command level query syntax is different from the one used in rulebases which might introduce some confusion to non-expert users.
- Restricted number of Prolog external predicates is supported. Better integration is necessary.

The next developments of the MWeb framework will include:

- Support of RIF and XML Schema datatypes.
- Direct translation of (Extended) RDF Schema ontologies in XML and NTriples serialization formats, including automatic support and renaming of blank nodes.
- Rulebases specified in Rule Interchange Format, both in presentation and normative XML syntax.

- Support of the basic OWL2 profiles, namely OWL2 RL which has a specification on RIF.
- Mechanisms for ontology mapping, in particular mapping of constants in different rulebases.
- Development of a paraconsistent semantics for MWebAS.

10. Library Predicates

Compilation

`mwebCompileModule( +RulebaseName )`
Compiles a rulebase. The RulebaseName is used to generate the file names of the interface and logic program files by encoding it and appending the suffixes, '.mw' and '.rb', respectively. Encoding escapes all chars except for letters, digits, HYPHEN-MINUS ("-") and LOW_LINE ("_"). The `mwebCompileModule/2` is then called for performing the compilation.

`mwebCompileModule( +InterfaceFile, +RulesFile )`
A prolog module will be generated and compiled. The name of the Prolog file is obtained from the rulebase declaration in the interface source file. The Prolog file has .pl extension and filename obtained by escaping all chars except for letters, digits, HYPHEN-MINUS ("-") and LOW_LINE ("_") of the rulebase IRI.

Loading

`mwebLoadModule( +RulebaseIRI )`
Loads a given rulebase identified by RulebaseIRI into the MWeb shell. The corresponding Prolog file with encoded RulebaseIRI filename and .pl extension must be in XSB's default search path.

`mwebUnloadModule( +RulebaseIRI )`
Removes the rulebase identified with RulebaseIRI from the MWeb shell. This can be used when local predicates have been defined more than once.

`mwebLoadModules( +ListOfRulebaseIRIs )`
Calls `mwebLoadModule/1` for each element in the argument list. Also accepts a single RulebaseIRI as in `mwebLoadModule/1`.

`mwebLoadAllModules( +RulebaseIRI )`
Convenience predicate to load recursively all the modules **EXPLICITLY** appearing in uses rulebase lists, starting from RulebaseIRI and following the rulebases' uses lists. If a rulebase has been previously loaded then it is ignored.

Declarations

`prefix/1`
Declares a set of prefixes to be used in the MWeb shell. The prefix command expects a comma separated list of terms Prefix='IRIRef'. The prefixes occurring in any loaded rulebase are **NOT** declared in the shell.

`vocabulary/1`
Declares extra vocabulary at the global level, visible to all or to a specific rulebase. It expects a comma separated list of constants with an optional @ Rulebase.

Querying

`mwebQuery( +FreeVars, +Query, +Semantics, +Regime )`
This is the main predicate for querying the MWeb system. This predicate executes Query with free variables listed in FreeVars under Semantics in mode Regime. The use of each argument is now detailed:

> ***FreeVars***: in some queries it might be necessary to list the free variables in FreeVars to perform instantiation before querying in order to avoid floundering problems (calls of weak negation with non-ground arguments). Users may try first calling with an empty list and then include the necessary variables to avoid errors.
>
> ***Query***: the query to be executed. More details can be found in the previous sections.
>
> ***Semantics***: either the constant `wfs` or `asp`, selecting respectively MWebWFS or MWebAS semantics to be used. In order to query under MWebAS semantics, XSB system must be compiled with smodels support. It is always possible to query with MWebWFS semantics, which is the default mode.
>
> ***Regime***: selects the MWeb querying mode: `d` for definite, `o` for open, `c` for closed, or `n` for normal. The default mode is `n`.

`mwebQuery( +Query )`
The same as `mwebQuery( [], Query, wfs, n )`.

`mwebQuery( +FreeVars, +Query )`
The same as `mwebQuery( FreeVars, Query, wfs, n )`.

`mwebQuery( +FreeVars, +Query, +Semantics )`
The same as `mwebQuery( FreeVars, Query, Semantics, n )`.

`mwebSetErrorLevel( +Level )`
Flag to control runtime behaviour when call of a normal predicate is made from a non-normal predicate (definite, open or closed). Level can be one of: `error`, `msg`, or `none`.
If `error` is set (the default) a runtime error is thrown, if `msg` is set then a warning message is written in the console, otherwise the computation continues silently (level `none`).

Consistency Checking

`mwebCheckConsistency( +Semantics, +Regime )`
Checks global consistency of all loaded rulebases under Semantics and mode Regime. Semantics takes values `wfs` or `asp`, and Regime values `d`, `o`, `c`, or `n`. Notice that the rulebases might be inconsistent in one regime and consistent in other regimes.

`mwebCheckConsistency`
The same as `mwebCheckConsistency( wfs, n )`.

Model Construction

`mwebModel( Rulebase, +Semantics, +Regime, PredInd, ?Literal )`
Backtracks over all literals true in Rulebase constructed from PredInd=PredName/Arity under Semantics with mode Regime. If Rulebase is a variable, then it returns literals true in the global interpretation. If PredInd is a variable then it is returned all literals true of defined predicates in the provided rulebase.

`mwebModel( Literal )`
The same as `mwebModel( _, wfs, n, _, Literal )`.

`mwebModel(Rulebase, Literal )`
The same as `mwebModel( Rulebase, wfs, n, _, Literal )`.

`mwebModel(Rulebase, Semantics, Literal )`
The same as `mwebModel( Rulebase, Semantics, n, _, Literal )`.

`mwebModel(Rulebase, Semantics, Regime, Literal )`
The same as `mwebModel( Rulebase, Semantics, Regime, _, Literal )`.

11. Readings

Anastasia Analyti, Grigoris Antoniou, Carlos V. Damasio, *MWeb: A Principled Framework for Modular Web Rule Bases and its Semantics,* Submitted for publication, 2010.

Anastasia Analyti, Grigoris Antoniou, Carlos V. Damasio, *A Formal Theory for Modular ERDF Ontologies,* Proceedings of Third International Conference Web Reasoning and Rule Systems (RR 2009), pp. 212-226, Chantilly, VA, USA, October 2009, Springer-Verlag.

Anastasia Analyti, Grigoris Antoniou, Carlos V. Damasio, *A Principled Framework for Modular Web Rule Bases and its Semantics,* Procs. of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008), pp. 390-400, Australia, September 2008, AAAI Press.

Anastasia Analyti, Grigoris Antoniou, Carlos V. Damasio, Gerd Wagner, *Extended RDF as a Semantic Foundation of Rule Markup Languages,* Journal of Artificial Intelligence Research (JAIR), 32, pp. 37-94, 2008, AAAI Press.

Carlos V. Damasio, Anastasia Analyti, Grigoris Antoniou, Gerd Wagner, *Supporting Open and Closed World Reasoning on the Web,* Procs. of 4th Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR 2006), Budva, Montenegro, pp. 149-163, June 2006, Springer-Verlag .