

This module is an input filter for the ProCom system. It implements the translation of multi-modal formulae into first order formulae of a constraint logic with terms representing the paths. Together with the translation, several techniques are used in order to increase the efficiency of the prover. This was achieved by decreasing the complexity of the terms and formulae. The main techniques are antiprenexing and a generalised skolemisation. All the details are discussed in an accompanying paper (which is still to be written though).

The formulae are expected to be in an input file and written in a language to be defined. This filter produces a matrix from the formula which in turn is an input for the ProCom system and a descriptor set for the particular problem.

After the translation, we normalised the path terms mentioned above using an algorithm implemented by Gilbert Boyreau, Caen University, France. Due to the strategy of Prolog of liking consecutive clauses, we had to put all translated clauses in a database and to write them out at the very end of the filter predicate.

The translation is carried out according to a suggestion in

Françoise Clerin-Debart

*Théories équationnelles et de contraintes pour la démonstration automatique en logique multi-modale*

PhD Thesis, Caen University, France, 1992

Modal axioms can be specified in several forms, which will be explained below. These axioms will be either coded as clauses or descriptors will be generated. In the latter case, the prover will be forced to use the generated set of descriptors rather than the standard descriptors. The choice of which axioms to code as descriptors and which to leave as clauses has been made from experimenting. Thus, it is not guaranteed that always the best choice will be made. Maybe that in later versions a switch will be provided to determine the degree of how many of the axioms to generate as descriptors. Generally, it can be said that axioms, which are Prolog facts, will be left as clauses, and thus in the matrix, and all other axioms will be coded as descriptors.

## The input language

Here, we want to present the input language. For more details, we refer the reader to the exact definition of the operators below. Table 1 provides an overview of all the defined operators.

The exact precedence of the operators can be seen in the listing of the module below. It can be seen, that disjunction and conjunction are defined as infix operators and all the other are prefix operators. The arity of 2 for the quantifiers seems to be confusing but we have to indicate the variable that we are quantifying and the formula over which we are quantifying. The same applies to the modal operators of arity 2. However, the first argument for this operator is not a variable but a sort of the modality. The modal operators of arity 1 are merely thought for the monomodal case, where no sort specification is required.

Three more syntactical remarks have to be made:

Sign	Notation	Arity	Example
$\wedge$	<code>_ and _</code>	2	<code>p(X) and q(Y)</code>
$\vee$	<code>_ or _</code>	2	<code>p(X) or q(Y)</code>
$\neg$	<code>not _</code>	1	<code>not p(X)</code>
$\rightarrow$	<code>_ implies _</code>	2	<code>p(X) implies q(Y)</code>
$\leftrightarrow$	<code>_ equivalent _</code>	2	<code>p(X) equivalent q(Y)</code>
$\forall$	<code>forall _ : _</code>	2	<code>forall X : p(X)</code>
$\exists$	<code>exists _ : _</code>	2	<code>exists X : p(X)</code>
$\Box$	<code>box _</code>	1	<code>box p(X)</code>
$\Box$	<code>box _ : _</code>	2	<code>box a : p(X)</code>
$\Diamond$	<code>diamond _</code>	1	<code>diamond p(X)</code>
$\Diamond$	<code>diamond _ : _</code>	2	<code>diamond a : p(X)</code>

Table 1: The input language of the filter `tom`

1. During the translation, constraints for the variables ranging over labels are introduced. These constraints are named `$Rmod_` and a term representing the sort of the constraint. This term is usually a lower case letter but can be anything.

Hence, the user is not advised to use function or predicate symbols with the same name. Otherwise, the system will be extremely puzzled (and the user by the result).

2. It has been noted in the literature, that rigid and flexible symbols (this refers to function, predicate, and constant symbols equally) have to be distinguished syntactically. Our choice was to prepend such symbols with a `$R`.
3. As any term beginning with a Dollar sign is interpreted as a variable by Prolog, the terms from above have to be quoted in single quotes.

The choice we made in the syntax was totally arbitrary, of course.

## Controlling the behaviour

There are six `Protop` options to control the behaviour of the filter `tom`. These options are given in table 2.

The default values for the options are underlined and will be overridden by specifications in the actual problem. The options will be explained in due course.

There are three way sof coding problem specific knowledge:

1. Using the command `# modal_standard.theory(theory_name,sort)`, a whole theory can be specified.

All options are set according to the particular requirements of this logic. For instance, a suitable unification algorithm will be specified to be used.

	option name	currently recognised values
1.	'Tom:method'	constraints, <u>inference</u>
2.	'Tom:merging_predicate'	off, <u>merge_clauses</u>
3.	'Tom:normal_form'	off, <u>negation_normal_form</u>
4.	'Tom:special_path_term'	off, <u>normalize_path</u>
5.	'Tom:special_unification'	off, <u>normalize_unify_a1</u>
6.	'Tom:log_file'	off, <u>on</u>
(7.)	('Tom:theory_optimization')	(off, <u>on</u> )

Table 2: Options for `tom` in `Protop`

At the moment, only the logic KD and S4 are recognised. This is due to the fact that no more implementation of unification algorithms were available. However, it is very easy to extend the framework provided for more logics. It is aimed at providing a proof tool for all normal modal logics or even certain temporal logics.

The user is encouraged to use the above command whenever possible.

**Example:** `# modal_standard_theory(s4,a)`

This specifies the accessibility relation `a` as being reflexive, transitive, serial, and normal.

2. The command `# modal_axiom_schema(schema,sort)` specifies one particular modal axiom schema. As there is no global overview of all specified axioms, a unification and/or a preprocessing of the path terms prior to unification is to be set by the options `'Tom:special_unification'` and `'Tom:special_path_term'` respectively.

This variant requires more care than the first one but is certainly more suitable for own experiments.

**Example:** `# modal_axiom_schema(transitive,s).`

This specifies the accessibility relation `s` as being transitive. Recognised values are `interaction`, `transitive` (4 alternatively), `euclidean` (5 alternatively), `reflexive` (t alternatively), and `total`.

3. The third command `# modal_axiom_formula(formula)` is used to specify a formula as being an axiom.

The user can specify any formula in form of a Prolog clause. However, he has to take care of the conventions from above regarding the names of the constraint predicates, variables, etc. This clause is put straight into the matrix. No checks of correctness or whatever will be carried out! The responsibility is entirely up to the user. Therefore it is not encouraged to use this option.

**Example:**        `# modal_axiom_formula('$Rmod_s'(A + B) :-  
'$Rmod_s'(A), '$Rmod_s'(B)).`

This example simply says that you have a label  $A + B$  of type  $s$  if you have labels  $A$  and  $B$  of type  $s$ .

The seventh option, `'Tom:theory_optimization'`, is only defined if the method `constraints` is chosen in the option `'Tom:method'`. The value of this option is paramount to any further treatment of the constraints.

The options `'Tom:merging_predicate'`, `'Tom:normal_form'`, `'Tom:special_path_term'`, and `'Tom:special_unification'` contain as values names of predicates if they are not set to `off`. The convention will be explained by the example of the option `'Tom:normal_form'`. If this option is set to `normalize_path`, it is expected that a file `normalize_path.pl` can be found. This file must contain a predicate `normalize_path`. The arity of this predicate is 2. The table 3 provides an overview of the requires arities of the predicates in the options.

option name	arity of the predicate
<code>'Tom:merging_predicate'</code>	3
<code>'Tom:normal_form'</code>	2
<code>'Tom:special_path_term'</code>	2
<code>'Tom:special_unification'</code>	2

Table 3: Arities for predicates specified by the options

The last argument of these predicates is the output argument. When merging clause lists, we need two input arguments for this predicate. All other options need one input argument only.

## Examples

Having introduced the input language and discussed syntactic restrictions, we want to give examples of how to use the filter. For instance, the modal logic formula  $\Box_a(w(a) \vee \neg w(a))$  would be written as `box a : ( w('$Ra') or not w('$Ra'))`.

It can be seen that the letter **a** carries two meanings which is not necessarily the case in the formulation of a problem. On the one hand, the letter **a** behind the box operator stands for the sort of the modal operator. On the other hand, it stands for a logical constant. As this constant is rigid, it is written as `'$Ra'`. Otherwise, the usual conventions of bracketing apply.

As it was mentioned earlier, there are different ways of specifying the modal axioms:

1. `# modal_standard_theory(theory_name,sort)` is used to specify a whole standard theory for a sort.

2. `# modal_axiom_schema(schema,sort)` is used to specify a particular axiom schema to be used.
3. `# modal_axiom_formula(formula)` is used to specify a formula as being an axiom.

Any other statement beginning with a `#` sign are not understood by the filter. Hence, it is assumed that this statement is not intended for the filter and the entire line will be put into the output file as it came.

It is important to write the full stop at the end of any line - we are writing Prolog after all. Comments are to be in lines beginning with `%`, the usual Prolog comment sign.

Having introduced all the syntactic details, we can give an example. We produce a stripped-down version of the wise men puzzle, which is a classical example for constraint logic programming in this context.

The following formulae are the facts known:

$$\begin{aligned} \Box_a \Box_b w(a) \\ \neg \Box_a w(a) \end{aligned}$$

Moreover, the relation  $b$  is known to be reflexive.

The input file looks as follows:

```
box a : box b : w('$Ra').
not (box a: w('$Ra')).

# modal_axiom_schema(reflexive,b).
```

The result of the translation using the `constraints` method is:

```
w(A + B, '$Ra')
    //: '$Rmod_b'(B) , '$Rmod_a'(A) .

-(w('$kolemPath1'(0), '$Ra')).

# begin(constraint_theory).
'$Rmod_a'('$kolemPath1'(0)).
'$Rmod_b'(0).
# end(constraint_theory).
```

...

The second method, `inference`, gave the following result:

```
'$Rmod_a'('$kolemPath1'(0)).

'$Rmod_b'(0).
```

```
w(B + A, '$Ra') :-
    '$Rmod_a'(B),
    '$Rmod_b'(A).

-(w('$kolemPath1'(0), '$Ra')).
```

The descriptor set for this particular problem is:

```
:- module('wise2c.x_descriptors').

:- compile(library(capri)).
:- lib(literal).
:- lib(matrix).
make_pred(C, B, A) :-
    C =.. [E, D],
    functor(D, G, F),
    functor(H, G, F),
    (
        E = --
    ->
        I = ++
    ;
        (
            E = ++
        ->
            I = --
        )
    ),
    A =.. [I, H],
    B =.. [E, H].

look_for_entry(D, C, B, A) :-
    make_pred(D, C, B),
    'Clause'(B, E, A).

descriptor((proof(reduction(C, B -> A))),
    template(B, goal),
    call(make_pred(B, D, A)),
    template(A, path(C)),
    constructor(normalize_unify_a1(B, D))).

descriptor((proof(connection(C, B -> A))),
    template(B, goal),
    call(look_for_entry(B, D, A, C)),
    constructor(normalize_unify_a1(B, D)),
    template(A, extension(C))).
```

The sample run for this example is as follows:

```

upwards> protop

--- Welcome to ProTop (1.51) ---

ProTop -> input_filter = tom.
ok.
ProTop -> prove('wise2.easy').
tom_ops.pl compiled traceable 0 bytes in 0.02 seconds
./tom.pl    compiled traceable 21548 bytes in 0.35 seconds
merge_clauses.pl compiled traceable 1712 bytes in 0.03 seconds
negation_normal_form.pl compiled traceable 6428 bytes in 0.07 seconds
tom_inference.pl compiled traceable 7000 bytes in 0.08 seconds
normalize_path.pl compiled traceable 1708 bytes in 0.02 seconds
% Input filter tom: 350 ms
4 clauses read in 67 ms
/u/home/procom/ProCom/capri.pl compiled traceable 0 bytes in 0.03 seconds
./wise2.easy_descriptors compiled traceable 2956 bytes in 0.08 seconds
--- ProCom: CaPrI module wise2.easy_descriptors loaded.

% All negative clauses are considered as goals.
complete_goals: 0 ms
connection_graph: 67 ms

Compile time: 1s 633 ms
...prover.pl compiled optimized 18868 bytes in 0.22 seconds
% Depth = 1
% Depth = 2
Runtime 33 ms

%| Proof with goal clause 4
%| connection
%| 3 - 1
%| --(w('$kolemPath1'(0), '$Ra')) -> ++(w('$kolemPath1'(0) + 0, '$Ra'))
%| | connection
%| | 1 - 1
%| | --('$Rmod_a'('$kolemPath1'(0))) -> ++('$Rmod_a'('$kolemPath1'(0)))
%| | connection
%| | 2 - 1
%| | --('$Rmod_b'(0)) -> ++('$Rmod_b'(0))
ok.
ProTop ->

```

The meaning of descriptors is not subject of this documentation but the reader is referred to the documentation of the entire ProCom system:

Gerd Neugebauer  
ProCom/CaPrI and the Shell ProTop — A User's Guide  
IMN, HTWK Leipzig, 1994