# Direct Resolution for Modal-like Logics

Carlos Areces Juan Heguiabehere
{carlos, juanh}@science.uva.nl

ILLC, Faculty of Science, University of Amsterdam

**Abstract.** In this paper we discuss the implementation of a hybrid logic theorem prover based on resolution, HyLoRes, which fuses first-order proving techniques with modal and hybrid reasoning. The focus is on implementation details, and on how we adapted first-order methods to the hybrid environment.

## 1 Introduction

Designing resolution methods that can directly (without translation into large background languages) be applied to modal logics, received some attention in the late 1980s and early 1990s; see for example [14, 13]. Given the simplicity of propositional resolution and the fact that modal languages are sometimes viewed as "simple extensions of propositional logic," we might expect modal resolution to be as simple and elegant. However, direct resolution for modal languages proved to be a difficult task. Intuitively, in basic modal languages the resolution rule has to operate *inside* boxes and diamonds to achieve completeness. This leads to more complex systems, less elegant results, and poorer performance, ruining the "one-dumb-rule" spirit of resolution. In [2] a resolution calculus for hybrid logics addressing these problems was introduced: the hybrid machinery is used to "push formulas out of modalities" and in this way, feed them into a simple and standard resolution rule.

We cannot give a proper introduction to hybrid logics in this paper. See the Hybrid Logic site at `http://www.hylo.net` for an introduction to the topic and a broad on-line bibliography. We intend instead to describe HyLoRes, an automated theorem prover based on the calculus introduced in [2], with special emphasis on implementation details. [3] provides a description of HyLoRes focused on the theoretical aspects involved in its development.

*Structure of the paper.* In Section 2 we give the syntax and semantics of the hybrid logics we are going to work with, the logics $\mathcal{H}(@)$ and $\mathcal{H}(@, \downarrow)$. In Section 3 we briefly introduce the calculus used in HyLoRes. In Section 4 we describe the algorithm we used as a starting framework. In Section 5 we discuss the language we chose to do the implementation, and explain how to run HyLoRes. Section 6 discusses implementation details, such as data structures and optimizations. Section 7 shows some ongoing testing. Finally, Section 8 shows the next steps planned in the development of HyLoRes.

## 2 Hybrid Languages

*Syntax.* Let $\mathsf{REL}$ be a countable set of *relational symbols*, $\mathsf{PROP}$ a countable set of *propositional variables*, $\mathsf{NOM}$ a countable set of *nominals*, and $\mathsf{SVAR}$ a countable set of *state variables*. We assume that these sets are pairwise disjoint. We call $\mathsf{SSYM} = \mathsf{NOM} \cup \mathsf{SVAR}$ the set of *state symbols*, and $\mathsf{ATOM} = \mathsf{PROP} \cup \mathsf{NOM} \cup \mathsf{SVAR}$ the set of *atoms*. The well-formed formulas of the hybrid language $\mathcal{H}(@, \downarrow)$ in the signature $\langle \mathsf{REL}, \mathsf{PROP}, \mathsf{NOM}, \mathsf{SVAR} \rangle$ are

$$\mathsf{FORMS} := \top \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid [R]\varphi \mid @_s\varphi \mid \downarrow x.\varphi,$$

where $a \in \mathsf{ATOM}$, $x \in \mathsf{SVAR}$, $s \in \mathsf{SSYM}$, $R \in \mathsf{REL}$ and $\varphi, \varphi_1, \varphi_2 \in \mathsf{FORMS}$. Note that all types of atomic symbol (i.e., proposition symbols, nominals and state variables) are *formulas*. Further, note that the above syntax is simply that of ordinary (multi-modal) propositional logic extended with clauses for $@_s\varphi$ and $\downarrow x.\varphi$. Finally, the difference between nominals and state variables is simply this: nominals cannot be bound by $\downarrow$, whereas state variables can.

The notions of *free* and *bound* variable are defined as in first-order logic, with $\downarrow$ as the only binding operator. A *sentence* is a formula containing no free state variables.

The basic hybrid language is $\mathcal{H}$, basic modal logic extended with nominals. Further extensions are usually named by listing the added operators; we are interested in the logics $\mathcal{H}(@)$ and $\mathcal{H}(@, \downarrow)$.

*Semantics.* A (hybrid) *model* $\mathcal{M}$ is a triple $\mathcal{M} = \langle M, \{R_i\}, V \rangle$ such that $M$ is a non-empty set, $\{R_i\}$ is a set of binary relations on $M$, and $V : \mathsf{PROP} \cup \mathsf{NOM} \to Pow(M)$ is such that for all nominals $i \in \mathsf{NOM}$, $V(i)$ is a singleton subset of $M$.

An *assignment* $g$ for $\mathcal{M}$ is a mapping $g : \mathsf{SVAR} \to M$. Given an assignment $g$, we define $g^x_m$ (an *$x$-variant of $g$*) by $g^x_m(x) = m$ and $g^x_m(y) = g(y)$ for $x \neq y$. Assignments are not needed when dealing with $\mathcal{H}(@)$.

Let $\mathcal{M} = \langle M, \{R_i\}, V \rangle$ be a model, $m \in M$, and $g$ an assignment. For any atom $a$, let $[V, g](a) = \{g(a)\}$ if $a$ is a state variable, and $V(a)$ otherwise. Then the *satisfiability relation* is defined as follows:

$$\begin{array}{lcll}
\mathcal{M}, g, m \Vdash & \top & \text{always} & \\
\mathcal{M}, g, m \Vdash & a & \text{iff} & m \in [V, g](a),\ a \in \mathsf{ATOM} \\
\mathcal{M}, g, m \Vdash & \neg\varphi & \text{iff} & \mathcal{M}, g, m \nVdash \varphi \\
\mathcal{M}, g, m \Vdash & \varphi_1 \wedge \varphi_2 & \text{iff} & \mathcal{M}, g, m \Vdash \varphi_1 \text{ and } \mathcal{M}, g, m \Vdash \varphi_2 \\
\mathcal{M}, g, m \Vdash & [R]\varphi & \text{iff} & \forall m'.(R(m, m') \implies \mathcal{M}, g, m' \Vdash \varphi) \\
\mathcal{M}, g, m \Vdash & @_s\varphi & \text{iff} & \mathcal{M}, g, m' \Vdash \varphi, \text{ where } [V, g](s) = \{m'\} \\
\mathcal{M}, g, m \Vdash & \downarrow x.\varphi & \text{iff} & \mathcal{M}, g^x_m, m \Vdash \varphi.
\end{array}$$

We write $\mathcal{M}, g \Vdash \varphi$ iff for all $m \in M$, $\mathcal{M}, g, m \Vdash \varphi$; and $\mathcal{M} \Vdash \varphi$ iff for all $g$, $\mathcal{M}, g \Vdash \varphi$. These notions extend to sets of formulas in the standard way. A formula $\varphi$ is *satisfiable* if there is a model $\mathcal{M}$, an assignment $g$ on $\mathcal{M}$, and a world $m \in M$ such that $\mathcal{M}, g, m \Vdash \varphi$. A formula $\varphi$ is *valid* if for all models $\mathcal{M}$, $\mathcal{M} \Vdash \varphi$.

## 3 The Resolution Calculus

Briefly, the resolution calculus implemented by HyLoRes is as follows. Define the following rewriting procedure $nf$ on formulas of $\mathcal{H}(@,\downarrow)$. Let $\varphi$ be a formula in $\mathcal{H}(@,\downarrow)$, $nf(\varphi)$ is obtained by repeated application of the rewrite rules $nf$ until none is applicable:

$$\begin{array}{rcl} \neg @_t\psi & \overset{nf}{\rightsquigarrow} & @_t\neg\psi \\ \neg\downarrow x.\psi & \overset{nf}{\rightsquigarrow} & \downarrow x.\neg\psi \\ \neg\neg\psi & \overset{nf}{\rightsquigarrow} & \psi \end{array}$$

Clauses are sets of formulas in this normal form. To determine the satisfiability of a sentence $\varphi \in \mathcal{H}(@,\downarrow)$ we first notice that $\varphi$ is satisfiable iff $@_t\varphi$ is satisfiable, for a nominal $t$ not appearing in $\varphi$. Define the clause set *ClSet* corresponding to $\varphi$ to be $\mathit{ClSet}(\varphi) = \{\{@_t nf(\varphi)\}\}$. Next, let $\mathit{ClSet}^*(\varphi)$ – the saturated clause set corresponding to $\varphi$ – be the smallest set containing $\mathit{ClSet}(\varphi)$ and closed under the following rules.

$$(\wedge)\ \frac{Cl \cup \{@_t(\varphi_1 \wedge \varphi_2)\}}{\begin{array}{l} Cl \cup \{@_t\varphi_1\} \\ Cl \cup \{@_t\varphi_2\} \end{array}} \qquad (\vee)\ \frac{Cl \cup \{@_t\neg(\varphi_1 \wedge \varphi_2)\}}{Cl \cup \{@_t nf(\neg\varphi_1), @_t nf(\neg\varphi_2)\}}$$

$$(\mathrm{RES})\ \frac{Cl_1 \cup \{@_t\varphi\} \quad Cl_2 \cup \{@_t\neg\varphi\}}{Cl_1 \cup Cl_2}$$

$$([R])\ \frac{Cl_1 \cup \{@_t[R]\varphi\} \quad Cl_2 \cup \{@_t\neg[R]\neg s\}}{Cl_1 \cup Cl_2 \cup \{@_s\varphi\}} \qquad (\langle R\rangle)\ \frac{Cl \cup \{@_t\neg[R]\neg\varphi\}}{\begin{array}{l} Cl \cup \{@_t\neg[R]\neg n\} \\ Cl \cup \{@_n nf(\neg\varphi)\} \end{array}},\ \text{for } n \text{ new.}$$

$$(@)\ \frac{Cl \cup \{@_t@_s\varphi\}}{Cl \cup \{@_s\varphi\}}$$

$$(\mathrm{SYM})\ \frac{Cl \cup \{@_t s\}}{Cl \cup \{@_s t\}} \quad (\mathrm{REF})\ \frac{Cl \cup \{@_t\neg t\}}{Cl} \quad (\mathrm{PARAM})\ \frac{Cl_1 \cup \{@_t s\} \quad Cl_2 \cup \{\varphi(t)\}}{Cl_1 \cup Cl_2 \cup \{\varphi(t/s)\}}$$

**Fig. 1.** Resolution calculus for the logic $\mathcal{H}(@)$

The computation of $\mathit{ClSet}^*(\varphi)$ is in itself a sound and complete algorithm for checking satisfiability of $\mathcal{H}(@)$, in the sense that $\varphi$ is unsatisfiable if and only if the empty clause $\{\}$ is a member of $\mathit{ClSet}^*(\varphi)$ [2]. To account for hybrid sentences using $\downarrow$ we need only extend the calculus with the rule

$$(\downarrow)\ \frac{Cl \cup \{@_t\downarrow x.\varphi\}}{Cl \cup \{@_t\varphi(x/t)\}}.$$

The full set of rules is a sound and complete calculus for checking satisfiability of sentences in $\mathcal{H}(@,\downarrow)$.

## 4 The Given Clause Algorithm

HyLoRes implements a version of the "given clause" algorithm [22] shown in Fig. 2. The implementation preserves the soundness and completeness of the calculus introduced above, and ensures termination for $\mathcal{H}(@)$.

```
input: init: set of clauses
var: new, clauses, inuse: set of clauses
var: given: clause

clauses := {}; inuse := {}; new := normalize(init)
if {} ∈ new then return "unsatisfiable"
clauses := computeComplexity(new)
while clauses ≠ {} do
    {* Selection of given clause *}
    given := select(clauses); clauses := clauses – {given}
    {* Inference *}
    new := infer(given, inuse); new := normalize(new)
    if {} ∈ new then return "unsatisfiable"
    {* Subsumption deletion *}
    new := simplify(new, inuse ∪ clauses)
    inuse := simplify(inuse, new)
    clauses := simplify(clauses, new)
    {* Initialization for next cycle *}
    if notRedundant(given) then
        inuse := inuse ∪ {given}
    clauses := clauses ∪ computeComplexity(new)
return "satisfiable"
```

**Fig. 2.** Structure of the given clause algorithm

A brief explanation of the functions in Fig. 2:

- *normalize(A)* applies *nf* to formulas in $A$ and handles trivial tautologies and contradictions.
- *computeComplexity(A)* determines length, modal depth, number of literals, etc. for each of the formulas in $A$; these values are used by *select* to pick the given clause.
- *infer(*given*,A)* applies the resolution rules to the given clause and each clause in $A$. If the rules $(\wedge)$, $(\vee)$, $(\langle R\rangle)$ or $(\downarrow)$ are applicable, no other rule is applied as the clauses obtained as conclusions by their application subsume the premises.

– *simplify(A,B)* performs subsumption deletion, returning the subset of $A$ which is not subsumed by any element in $B$.
– *notRedundant(*given*)* is true if none of the rules ($\wedge$), ($\vee$), ($\neg[R]$) or ($\downarrow$) was applied to given.

# 5 Implementation

## 5.1 Programming Language

HyLoRes is implemented in Haskell (ca. 3500 lines of code), and compiled with the Glasgow Haskell Compiler (GHC) Version 5.04. We use Happy 1.13 to generate the parser. GHC produces fairly efficient C code which is afterward compiled into an executable file. Thus, users need no additional software to use the prover. The HyLoRes site (`http://www.illc.uva.nl/~carlos/HyLoRes`) provides executables for Solaris (tested under Solaris 8) and Linux (tested under Red Hat 7.0 and Mandrake 8.2).

The original Haskell code is also made publicly available under the GPL license [8] (the code, though, is still unstable, being under active development). We will soon provide also the intermediate C source which could then be compiled under a wider range of platforms. By making all code publicly available we aim to encourage independent development.

## 5.2 How to Run HyLoRes

HyLoRes can be run from the command line, in the following manner:

```
hylores -f filename [-o order | -t timeout | -sr | -r | -s]
```

where:

– `filename` is the input file.
– `order` is the ordering precedence for selection of the given clause:
  • `s`: Selects the clause with the least number of formulas.
  • `v`: Selects the clause with the greatest total number of propositional variables.
  • `d`: Selects the clause for which the maximum modal depth is the smallest.
  • `p`: Selects the clause for which the newest created nominal (as per the ($\neg[R]$) rule) is oldest.
– `timeout` is the time limit in seconds before the prover will stop.
– `-sr` selects direct (non-ordered) resolution.
– `-r` enables output of the rules applied and their results.
– `-s` enables output of the clause sets.

A typical run of HyLoRes is as follows:

**Input file:**

```
begin
!((down (x1 dia (x1 & p1) )) -> p1)
end
```

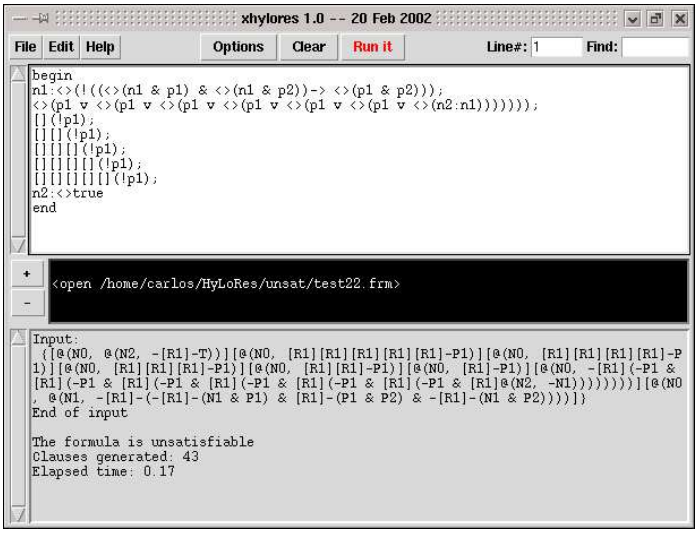
**Execution:**

```
(carlos@guave 149) hylores -f test.frm -r
Input:
   {[@(N0, (-P1 & Down(X1, -[R1]-(P1 & X1))))]}
End of input

Given: (1, [@(N0, (-P1 & Down(X1, -[R1]-(P1 & X1))))])
CON: {[@(N0, -P1)][@(N0, Down(X1, -[R1]-(P1 & X1)))]}
Given: (2, [@(N0, -P1)])
Given: (3, [@(N0, Down(X1, -[R1]-(P1 & X1)))])
ARR: {[@(N0, -[R1]-(P1 & N0))]}
Given: (4, [@(N0, -[R1]-(P1 & N0))])
DIA: {[@(N-2, (P1 & N0))][@(N0, -[R1]-N-2)]}
Given: (5, [@(N-2, (P1 & N0))])
CON: {[@(N-2, P1)][@(N-2, N0)]}
Given: (6, [@(N-2, N0)])
PAR (0,-2): {[@(N-2, (P1 & N-2))][@(N-2, -[R1]-(P1 & N-2))]
[@(N-2, Down(X1, -[R1]-(P1 & X1)))][@(N-2, -P1)]
[@(N-2, (-P1 & Down(X1, -[R1]-(P1 & X1))))]}
Given: (7, [@(N-2, P1)])
Given: (8, [@(N-2, -P1)])
RES: (7, [])

The formula is unsatisfiable
Clauses generated: 11
Elapsed time: 0.0
```

In addition to HyLoRes, a graphical interface called xHyLoRes implemented in Tcl/Tk was developed. It uses HyLoRes in the background and provides full file access and editing capabilities, and a more intuitive control of the command line parameters of the prover, in the manner of Spin/XSpin [19]. A screenshot of xHyLoRes can be seen below.



# 6 The Gory Details

## 6.1 Data Structures

The design of HyLoRes is modular with respect to the internal representation of the different kinds of data. We have used the Edison package [16] (a library of efficient data types provided with GHC) to implement most of the data types representing sets. The basic data types we created are as follows.

*Formulas.* We took advantage of the possibility of defining recursive data types in Haskell, with the result that the data type definition closely resembles the definition given in Section 2:

```
data Formula
   = Taut | Nom  Int | Prop Int | Var  Int | Neg  Formula | Con [Formula]
     | At   Int Formula | Atv  Int Formula | Down Int Formula
     | Box  Int Formula
deriving (Ord, Eq)
```

The integers in the definition represent the different elements of their corresponding sets, i.e. `Nom 1` represents the element $n_1$ in the set NOM, and so on. Conjunctions are stored as the `Con` constructor plus a list of conjuncts, to allow for n-ary conjunctions.

*Clauses and Sets of clauses.* The algorithm deals with three main repositories of clauses: *clauses*, that holds the eligible candidates por processing; *inuse*, that holds the clauses which can interact with the given clause, and *new*, where the clauses that result from the application of the rules go. The different clause sets and their clauses have different access patterns and aggregate information and need a different data type for each. *clauses* uses the `UnbalancedSet` type provided by the Edison library which is specially optimized for search; as in every cycle the given clause has to be selected from it. The comparison of clause scores is given as the ordering function, so the given clause can be selected without having to examine the whole set. The elements of *clauses* are tuples containing the clause proper (represented also as an `UnbalancedSet`), a complexity measure which depends on the chosen order for clause selection, and the clause number.

In *new*, clauses are stored as `UnbalancedSet`s while *new* itself is a list of clauses, as all its elements have to be processed one by one in each cycle. *inuse* is a list of pairs composed of the clause number and a clause represented also as a list, as both clauses and formulas in clauses need to be accessed one by one in every cycle.

*State and Output Monads.* Functional programming does not allow for global variables or side effects; in a function, all input must be passed as an argument and all consequences must be part of the returned value. For some applications, this can result in functions having very long and unintuitive lists of arguments, and contrived output types. In Haskell, a particular data type called `monad` is

used to overcome this problem. The internal state of the given clause algorithm (the sets *clauses*, *inuse* and *new*, the data structures used for subsumption checking, the control information, etc) is represented as a combination of a state and an output monads [24]; the former provides transparent access to the internal state of the algorithm from the monadic functions that perform inference, while the latter handles all printing services with no need of further parameters in the function signatures. In addition, the use of monads allows the addition of further structure (hashing functions, etc.) to optimize search, with minimum re-coding. We have already experienced the advantages of the monad architecture as we have been able to test different data structures and improve the performance of some of the most expensive functions with great ease.

### 6.2 Optimizations

The first implementation of HyLoRes was very naïve and as a result was terribly inefficient. We then proceeded to adapt and apply well established first-order resolution optimizations to the hybrid environment, with encouraging results.

*Ordered resolution with selection.* HyLoRes actually implements a version of *ordered resolution with selection* [5], where the application of the (RES) and ([$R$]) rules are restricted to certain selected formulas in the clause. Ordered resolution with selection greatly reduces the size of the saturated set, preventing the generation of certain clauses, without compromising the completeness of the calculus. Interestingly, the (still unpublished) proof of completeness of ordered resolution with selection for $\mathcal{H}(@, \downarrow)$ closely follows the proof in [5], based on a step by step construction of a Herbrand model for any consistent input clause set. Once more, hybrid logics seem to provide the appropriate framework to merge first-order and modal ideas.

We are currently investigating the effect of different orders and selection functions on different inputs.

*Formula indexing.* The formulas are indexed using a mapping between formulas and integers, in which indexes for positive and negative occurrences of the same formula will be equal except for the sign. As the (RES) rule involves searching for complementary formulas, searching for clauses to resolve with is made more efficient by storing the clauses in *inuse* as ordered lists of the indexes. This indexing is much simpler than in the case of first-order, as clauses do not have free variables.

*Subsumption checking.* Whenever a clause $A$ follows from another clause $B$ in the database, $A$ is said to be subsumed by $B$, and can be ignored, reducing the search space but maintaining correctness. Finding out which clauses can be discarded is one of the – or perhaps "the" – most expensive operations in resolution based theorem provers [21]. HyLoRes uses a simple version of subsumption checking where a clause $C_1$ subsumes a clause $C_2$ if $C_1 \subset C_2$. Version 0.5 of the

prover implemented this test very inefficiently, checking the subset relation element by element, and clause by clause. In the latest prototype, a set-at-a-time subsumption checking algorithm which uses a clause repository structured as a trie [21] was implemented, with notorious improvements (See Section 7). We also noticed that while forward subsumption is essential, many times backward subsumption does not really make a difference. This is also the case for some first-order logic provers; see [23].

The clause repository is organized as a list of tries, in the following manner. The clauses are inserted and queried as ordered lists of integers. The repository is a list of tries, in which each node represents a formula and each path that ends in a leaf node represents a clause.

*Example 1.* The set of clauses

$$\{ \quad \{1,2,3\}, \quad \{1,2,4\}, \{1,2,8\}, \quad \{1,4,5\}, \quad \{1,4,7,8\},$$
$$\{1,4,7,9,10\}, \{2,3,9\}, \{2,7,9\}, \{2,7,8,10\}, \{2,7,8,11\} \ \}$$

is stored as shown in Fig. 3.

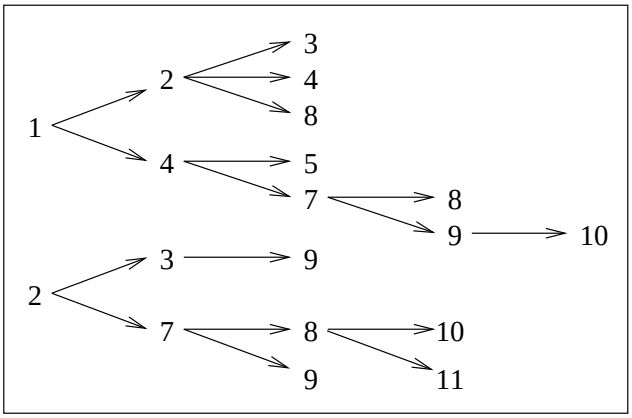



**Fig. 3.** Trie representation for a set of clauses

When inserting a clause, if its head is the root of any of the visible tries then we insert its tail into that trie, otherwise we add a branch to the current node and insert the clause there. In this way, all clauses are represented as a path from one of the root nodes to a leaf, so that all the clauses that are extensions of a particular path are stored as branching from it. The fact that the formulas in the clause are ordered gives us the possibility to optimize search, both by having a unique representation and by knowing when it will be useless to keep searching. The clause repository holds both the clauses in *inuse* and the ones in *clauses*, so as to check for (forward or backward) subsumption against just one set of

clauses, which also eliminates the cost of transferring clauses from one trie to the other when a clause is moved from *clauses* to *inuse*. Subsumption checking has then become very efficient, and indeed it brought a speed up of about two orders of magnitude to the prover.

In forward subsumption, the clauses in *new* are checked one by one for subsumption by the clauses in *inuse* or *clauses*, as follows: for each clause $C$ in *new*, for each of the visible tries $T_i$ in the repository, if the root of $T_i$ is in the checked clause, all the branches of $T_i$ are successively checked for the elements of the clause that are greater than the root. If we reach the end of any branch, then the clause is subsumed by the repository and the search stops. If we find any element not present in $C$, none of the clauses represented by the current path subsumes $C$ and we can proceed to the next trie. If the root of the next trie is greater than the maximum element in $C$, no match will be possible and the search ends.

In backward subsumption, the clauses in *new* are checked one by one for subsumption of the clauses in *inuse* or *clauses*, as follows: for each clause $C$ in *new*, for each of the visible tries $T_i$ in the repository whose root is less than or equal to the head of the clause (the smallest element), if the root of $T_i$ is equal to the head of $C$, we check the branches of $T_i$ for existence of the elements in the tail of $C$, and if the root of $T_i$ is less than the head of $C$ we check the branches of $T_i$ for existence of the whole clause. When we find a match for the last element of the clause, we know that all the paths that originate from $T_i$ are subsumed by the clause: we retrieve all of them, and examine the next trie. When we reach a $T_i$ with a root greater than the head of $C$, the search ends.

*Input analysis.* At this moment, HyLoRes performs a very simple analysis of its input. It checks for the presence of the $[R], \langle R\rangle, @$ and $\downarrow$ operators and for nominals in order to know which rules will need checking for applicability. For example, if the $\downarrow$ operator does not appear in the input, then the $(\downarrow)$ rule is switched off and never attempted. Most first-order provers perform a far more detailed analysis of the input and decide heuristics and settings on account of their findings.

*Application of the rules.* The rules are applied in such a way as to make the sets of clauses grow as slowly as possible. For example, the $(\neg\wedge)$ is checked first of all, and if it's applied then no other rule is applied, and also the given clause is not added to *inuse* (the antecedent and consequent clauses are equivalent, but it does not show in our implementation of subsumption checking). The same is true of $(\wedge)$. Then (RES) is applied, and the empty clause is searched for in the result before proceeding with the rest of the rules.

Another thing that helps pruning the search space is postponing the creation of new prefixes (by application of the $\langle R\rangle$ rule) until the clause set is saturated for the current set of prefixes. Whenever the $\langle R\rangle$ rule can be applied, the application is postponed until *clauses* is empty.

*Paramodulation.* We need some kind of paramodulation to handle nominals and @. We can once more take advantage from experience in first-order resolution here. In [4], Bachmair and Ganzinger develop in detail the modern theory of equational reasoning for first-order saturation based provers. Many of the ideas and optimizations discussed there can and should be implemented in HyLoRes. In the current version, paramodulation is done naïvely, the only "optimization" being the orientation of equalities so that we always replace nominals by nominals which are lower in a certain ordering.

## 7 Testing

During the development of HyLoRes, we made extensive use of test sets like those described in [6, 15] to evaluate the performance of the prover and guide design decisions. Some results are shown in Fig. 4. An important drawback of these test sets though, is that they only provide *purely modal* input. We are at the moment developing a random generator of hybrid formulas following the algorithm presented in [17], to test the performance of HyLoRes on hybrid input.

*Hand-tailored tests.* Fig. 4 a) represents a set of runs of the Balsiger, Heuerding and Schwendimann test set [6], with different criteria for selecting the given clause, and the description logic prover RACER [11], version 1-6r2, included as a reference. This set consists of nine groups of two sets of 21 problems. Each problem in a set is exponentially more difficult than the previous one, each group consists of one set of satisfiable problems and one set of unsatisfiable problems, and each group represents a different type of problem. The graph represents the number of instances solved by the prover in each set before the 100 seconds time limit elapsed. Most of these sets have become trivial for mature modal provers; some of them being solved during simplification. Nonetheless, it provided a quick way to evaluate the prover in the early stages.

*Random tests: Random Modal QBF test set.* Fig. 4 b) shows a run of several versions of HyLoRes and other provers over a very easy area of the Random Modal QBF test set [15]. The X axis represents the number of clauses in the original QBF formula, and the Y axis represents the average time for solving an instance, with 64 samples/datapoint. The problems range from being all satisfiable at the left, to being all unsatisfiable at the right. We benchmarked HyLoRes 0.5 (no formula indexing, no clause repository), HyLoRes 0.9 (formula indexing, clause repository, backward subsumption still using clause-at-a-time comparison) and HyLoRes 1.0 (now with backward subsumption using set-at-a-time comparison). We also ran SPASS v. 1.0.3 [25] with the standard translation to first-order logic, MSPASS v. 1.0.0t.1.3 [18], *SAT version 1.3 [20], and RACER v. 1-6r2 on this test, to compare with state-of-the art provers; in general the times for these provers only reflect start up times, as revealed by the absence of the easy-hard-easy pattern. This test set allowed us to gauge the progress of HyLoRes as we added optimizations to it, although QBF derived modal formulas have a very

rigid structure, which meant that a good performance on this test set was not a guarantee of good performance overall. See [12] for more on this test set.

*Random tests: Random Modal CNF test set.* This more recent test set [17] generates random modal CNF formulas directly. We ran the test for $C = 2.5$, $V = 3$ and $D = 1$; Fig. 4 c) represents median time elapsed as a function of (number of clauses/number of variables). The timeout value was 100 seconds: again, it was too easy for mature provers to compare their performances, while for HyLoRes there were a few timeouts in the hardest area. Fig. 4 d) plots the satisfiable/unsatisfiable fractions in the test we just described. There are zones of the plot in which the sum of the satisfiable and unsatisfiable fractions is less than 1; this is due to timeouts, as the sum represents the fraction of problems solved before the time limit.

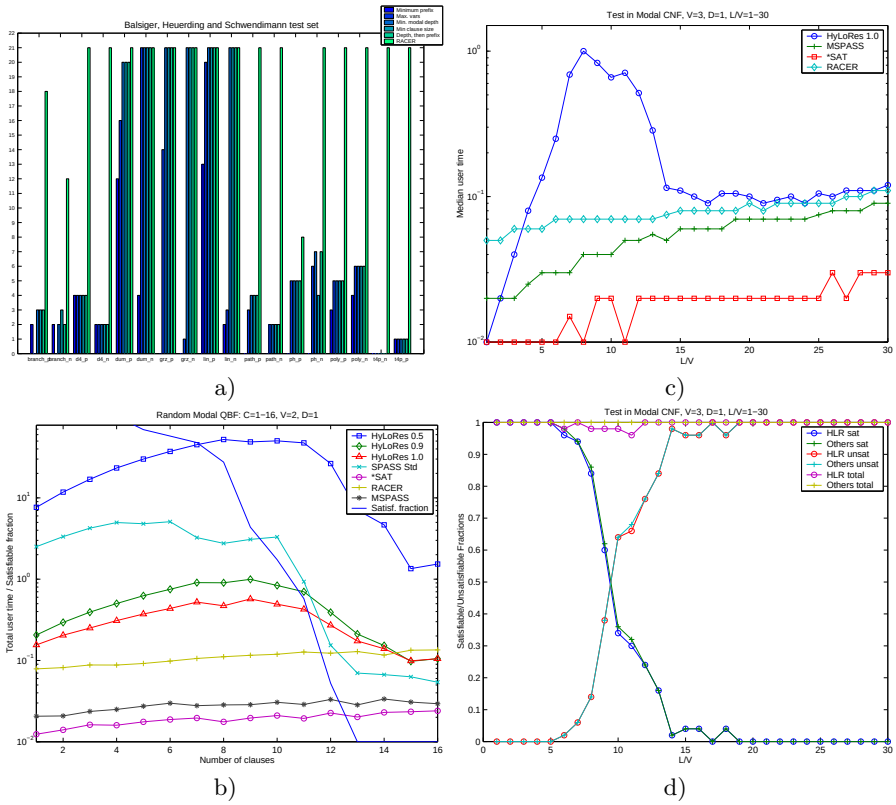



**Fig. 4.** Tests

## 8 Conclusions

The prototype is not yet meant to be competitive when compared with state of the art provers for modal-like logics like DLP, *SAT, MSPASS or RACER [7, 10]. On the one hand, the system is still in a preliminary stage of development (only very simple optimizations for hybrid logics have been implemented), and on the other hand the hybrid language and the languages handled by the other provers are related but different. $\mathcal{H}(@, \downarrow)$ is undecidable while the target languages of the other provers are decidable. And even when comparing the fragment $\mathcal{H}(@)$ for which HyLoRes implements a decision algorithm, the expressive powers are incomparable ($\mathcal{H}(@)$ permits free Boolean combinations of @ and nominals but lacks, for example, the limited form of universal modality available in the T-Box of DL provers [1]).

There certainly remain many things to try and improve in HyLoRes. The next steps in its development include

- a better treatment of paramodulation;
- support for the universal modality A [9] (which would allow us to perform inference in full Boolean knowledge bases of the description logic $\mathcal{ALCO}$);
- saving the saturated clause set, if any, for querying;
- and improve input analysis and heuristics.

But the main goal we pursued during the implementation of this prototype has been largely achieved: direct resolution can be used as an interesting, and perhaps even competitive, alternative to tableaux based methods for modal and hybrid logics.

**Acknowledgment.** C. Areces is supported by the NWO project # 612.069.006

## References

1. C. Areces. *Logic Engineering. The Case of Description and Hybrid Logics.* PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam, Amsterdam, The Netherlands, October 2000.
2. C. Areces, H. de Nivelle, and M. de Rijke. Resolution in modal, description and hybrid logic. *Journal of Logic and Computation*, 11(5):717–736, 2001.
3. C. Areces and J. Heguiabehere. HyLoRes: A hybrid logic prover based on direct resolution. *Proceedings of Advances in Modal Logic (AiML'02)*, 2002.
4. L. Bachmair and H. Ganzinger. Equational reasoning in saturation-based theorem proving. In *Automated deduction—a basis for applications, Vol. I*, pages 353–397. Kluwer Acad. Publ., Dordrecht, 1998.
5. L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.
6. P. Balsiger, A. Heuerding, and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. *Journal of Automated Reasoning*, 24(3):297–317, 2000.

7. E. Franconi, G. De Giacomo, R. MacGregor, W. Nutt, and C. Welty, editors. *Proceedings of the 1998 International Workshop on Description Logics (DL'98)*, 1998.
8. GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`.
9. V. Goranko and S. Passy. Using the universal modality: gains and questions. *Journal of Logic and Computation*, 2:5–30, 1992.
10. P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors. *Proceedings of the 1999 International Workshop on Description Logics (DL'99)*, 1999.
11. V. Haarslev and R. Möller. RACER System Description. `http://kogs-www.informatik.uni-hamburg.de/~race/`
12. J. Heguiabehere and M. de Rijke. The Random Modal QBF Test Set. In E. Giunchiglia and F. Massacci, editors *Proc. IJCAR Workshop on Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics*, 2001.
13. P. Enjalbert and L. Fariñas del Cerro. Modal Resolution in Clausal Form. *Theoretical Computer Science*, 65(1):1–33, 1989.
14. G. Mints. Resolution calculi for modal logics. *American Mathematical Society Translations*, 143:1–14, 1989.
15. F. Massacci. Design and results of the Tableaux-99 Non-Classical (Modal) Systems Comparison. In N. Murray, editor, *Proceedings of TABLEAUX'99*, volume 1617 of *LNAI*, pages 14–18. Springer-Verlag, 1999.
16. C. Okasaki. An Overview of Edison. *ICFP 2000 (Haskell Workshop)*, 2000. http://citeseer.nj.nec.com/okasaki00overview.html
17. P. Patel-Schneider and R. Sebastiani. A new very-general method to generate random modal formulae for testing decision procedures. *Journal of Artificial Intelligence Research*. Submitted.
18. R. Schmidt. MSPASS Home Page. `http://www.cs.man.ac.uk/~schmidt/mspass/`.
19. Spin – Formal Verification. `http://spinroot.com/spin/whatispin.html`.
20. A. Tacchella. *SAT Project Homepage. `http://www.mrg.dist.unige.it/~starsat/home.html`.
21. A. Voronkov. The anatomy of Vampire. *Journal of Automated Reasoning*, 15(2):237–265, 1995.
22. A. Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction. In R.Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning. IJCAR 2001*, number 2083 in LNAI, pages 13–28, Siena, 2001.
23. I.V. Ramakrishnan and R. Sekar and A. Voronkov. Term Indexing. In A. Robinson and A. Voronkov, editors, *Hanbook of Automated Reasoning*, pages 1853–1964, Elsevier Science, 2001.
24. P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in LNCS. Springer-Verlag, 1995.
25. C. Weidenbach. SPASS home page. `http://spass.mpi-sb.mpg.de/`.