HTab: A Terminating Tableaux System for Hybrid Logic

Guillaume Hoffmann, Carlos Areces

HTab: A Terminating Tableaux System for Hybrid Logic

Guillaume Hoffmann[1]

TALARIS
INRIA Lorraine
54602 Villers-lès-Nancy, France

Carlos Areces[2]

TALARIS
INRIA Lorraine
54602 Villers-lès-Nancy, France

Abstract

Hybrid logic is a formalism that is closely related to both modal logic and description logic. A variety of proof mechanisms for hybrid logic exist, but the only widely available implemented proof system, HyLoRes, is based on the resolution method. An alternative to resolution is the tableaux method, already widely used for both modal and description logics. Tableaux algorithms have also been developed for a number of hybrid logics, and the goal of the present work is to implement one of them.
In this article we present the implementation of a terminating tableaux algorithm for the basic hybrid logic. The performance of the tableaux algorithm is compared with the performances of HyLoRes and HyLoTab (a system based on a different tableaux algorithm).
HTab is implemented in the functional language Haskell, using the Glasgow Haskell Compiler (GHC). The code is released under the GNU GPL and can be downloaded from http://hylo.loria.fr/intohylo/htab.php.

Keywords: hybrid logic, tableaux method, theorem proving

1 Introduction

In this article we present the implementation of a terminating tableau algorithm for the basic hybrid logic $\mathcal{H}(@)$ described in [4]. The performance of the tableaux algorithm is compared with the performance of two other theorem provers for hybrid logics, HyLoRes (see [2]) and HyLoTab (see [5]). Some optimisations aimed at improving the behavior of the prover are also explored.

In section 2, we provide a brief introduction to hybrid logics, presenting the basic syntax and semantics for the hybrid language $\mathcal{H}(@)$. In section 3, we discuss

[1] Email: guillaume.hoffmann@loria.fr

[2] Email: carlos.areces@loria.fr

the main goals we have set for the HTab prover. In section 4, we present the rules of the tableaux method, their implementation, and some basic optimisations. In section 5, we list the result of preliminary testing. In the conclusion we see the perspectives for further developments of the prover.

2 The Hybrid Logic $\mathcal{H}(@)$

$\mathcal{H}(@)$ extends the basic modal language by adding nominals and satisfaction operator. The following definition gives the syntax and the semantic of this language.

Definition 2.1 Let $\mathsf{REL} = \{\Diamond_1, \Diamond_2, \ldots\}$ (*relational symbols*), $\mathsf{PROP} = \{p_1, p_2, \ldots\}$ (*propositional variables*) and $\mathsf{NOM} = \{i_1, i_2, \ldots\}$ (*nominals*) be disjoint and countable sets of symbols. Well formed formulas of the hybrid language $\mathcal{H}(@)$ in the signature $\langle \mathsf{REL}, \mathsf{PROP}, \mathsf{NOM} \rangle$ are given by the following recursive definition:

$$\mathsf{FORMS} ::= p \mid i \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \Diamond\varphi \mid @_i\varphi,$$

where $p \in \mathsf{PROP}$, $i \in \mathsf{NOM}$, $\Diamond \in \mathsf{REL}$ and $\varphi, \varphi_1, \varphi_2 \in \mathsf{FORMS}$.

A (hybrid) *model* $\mathcal{M}$ is a triple $\mathcal{M} = \langle M, (\Diamond^{\mathcal{M}})_{\Diamond \in \mathsf{REL}}, V \rangle$ such that M is a non-empty set, each $\Diamond^{\mathcal{M}}$ is a binary relation on M, and $V : \mathsf{PROP} \cup \mathsf{NOM} \rightarrow \wp(M)$ is such that for each nominal $i \in \mathsf{NOM}$, $V(i)$ is a singleton subset of M. We commonly write M for the domain of a model $\mathcal{M}$, and we call *states*, *worlds* or *points* the elements of M. Each $\Diamond^{\mathcal{M}}$ is an *accessibility relation*, and V is the *valuation.*

Let $\mathcal{M} = \langle M, (\Diamond^{\mathcal{M}})_{\Diamond \in \mathsf{REL}}, V \rangle$ be a model and $m \in M$. For each nominal $i \in \mathsf{NOM}$, let $[i]^{\mathcal{M}}$ be the state referred by i (i.e., for $i \in \mathsf{NOM}$, $[i]^{\mathcal{M}}$ is the unique $m \in M$ such that $V(i) = \{m\}$). Then, the *satisfaction relation* is defined as following:

$$
\begin{array}{lll}
\mathcal{M}, m \models p & \text{iif} & m \in V(p) \quad \text{for } p \in \mathsf{PROP} \\
\mathcal{M}, m \models i & \text{iif} & m = [i]^{\mathcal{M}} \quad \text{for } i \in \mathsf{NOM} \\
\mathcal{M}, m \models \neg\varphi & \text{iif} & \mathcal{M}, m \not\models \varphi \\
\mathcal{M}, m \models \varphi_1 \wedge \varphi_2 & \text{iif} & \mathcal{M}, m \models \varphi_1 \text{ and } \mathcal{M}, m \models \varphi_2 \\
\mathcal{M}, m \models \Diamond\varphi & \text{iif} & \text{exists a state } m' \text{ s.t. } \Diamond^{\mathcal{M}}(m, m') \text{ and } \mathcal{M}, m' \models \varphi \\
\mathcal{M}, m \models @_i\varphi & \text{iif} & \mathcal{M}, [i]^{\mathcal{M}} \models \varphi \quad \text{for } i \in \mathsf{NOM}.
\end{array}
$$

A formula φ is *satisfiable* if there is a model $\mathcal{M}$ and a world $m \in M$ such that $\mathcal{M}, m \models \varphi$. A formula φ is *valid* (notation: $\models \varphi$) if for all models $\mathcal{M}$, $\mathcal{M} \models \varphi$.

In [1], it is shown that the satisfiability problem for $\mathcal{H}(@)$ is decidable and PSPACE-complete.

3 Aims of HTab

The main goal behind HTab is to make available an optimised tableaux prover for hybrid logics, using algorithms that ensure termination. We ultimately aim to cover a number of frame conditions (i.e., reflexivity, symmetry, antisymmetry, etc.), as far as we can ensure termination. Moreover, we are interested in providing a range of inference services beyond satisfiability checking. For example, the current version of HTab includes model generation (i.e., HTab can generate a model from a saturated open branch in the tableau).

In this paper we report on version 1.1 of the prover. It is distributed under the GNU GPL, and the source code is available for download at http://hylo.loria.fr/intohylo/htab.php. For the moment, the prover only includes a few optimisations and can handle the basic modal logic $\mathcal{H}(@)$.

Even though other provers for languages similar to $\mathcal{H}(@)$ exists, HTab has a number of particularities that make it a potentially useful tool. We mention here some related provers, list their main characteristics and provide appropriate references. We will then comment on the main differences with HTab.

- RACER [7] implements a tableaux algorithm for a very expressive description logic ($\mathcal{ALCQHI}_{\mathcal{R}^+}$). It is highly optimised and very flexible. It implements state-of-the-art optimisations and heuristics, and provides inference services beyond satisfiability checking which are typical of description logic reasoners (building a concept taxonomy, retrieval, etc). The language $\mathcal{ALCQHI}_{\mathcal{R}^+}$ is incomparable with $\mathcal{H}(@)$. Intuitively, it has a restricted use of @, and nominals are not allowed.
- HyLoTab [5] is a tableaux based prover for the hybrid logics up to $\mathcal{H}(@, \Diamond^{-1}, \downarrow, \mathsf{A})$ ($\Diamond^{-1}$ is the inverse modality, $\downarrow$ is the 'bind-to-the-current-state' binder, and A is the universal modality). The prover can handle the reflexivity, transitivity and minimality frame conditions, and can generate a model from an open branch in the tableaux. The complete language $\mathcal{H}(@, \Diamond^{-1}, \downarrow, \mathsf{A})$ is undecidable (the $\downarrow$ binder is to blame), and hence, general terminating algorithms are not possible. But, unfortunately, the rules implemented by HyLoTab do not guarantee termination even for decidable sub-fragments of $\mathcal{H}(@, \Diamond^{-1}, \downarrow, \mathsf{A})$ like $\mathcal{H}(@, A)$ [3].
- HyLoRes [2] is a resolution based prover for the hybrid logics up to $\mathcal{H}(@, \Diamond^{-1}, \downarrow)$. The implemented algorithm is terminating for formulas in $\mathcal{H}(@, \Diamond^{-1})$, and does model generation, but it doesn't handle frame conditions. The prover actually performs resolution with order and selections functions, and different orders and selection functions can be specified. The complexity of the implemented algorithm is EXPTIME, even for fragments of $\mathcal{H}(@, \Diamond^{-1}, \downarrow)$ with lower complexity.

As we said above, HTab has particularities that differentiate it from each of the three provers we just mentioned. To start with, it handles the hybrid operators (@ and nominals) with no restrictions and it performs model generation. These two features distinguishes it from RACER. On the negative side, the current version of HTab has only a few optimisations, while RACER is a mature theorem prover that includes most state-of-the-art optimisation techniques. We aim to incorporate further optimisations (e.g., model caching) step by step, in future versions of the prover. HyLoTab is the system most similar to HTab, being both tableaux based provers for hybrid logic. Besides some technical issues (the way in which substitutions are handled in HyLoTab differs from the approach taken in HTab) the main difference is termination: one of the main aims of HTab is to always ensure that

[3] For instance, the formula $\neg p \wedge A(p \vee \Diamond(\neg p \wedge n))$ makes HyLoTab loop.

$$\frac{\sigma{:}(\varphi \wedge \psi)}{\sigma{:}\varphi, \sigma{:}\psi}\,(\wedge) \qquad \frac{\sigma{:}(\varphi \vee \psi)}{\sigma{:}\varphi \mid \sigma{:}\psi}\,(\vee)$$

$$\frac{\sigma{:}\Diamond\varphi}{\sigma{:}\Diamond\tau, \tau{:}\varphi}\,(\Diamond)^1 \qquad \frac{\sigma{:}\Box\varphi, \sigma{:}\Diamond\tau}{\tau{:}\varphi}\,(\Box)$$

$$\frac{\sigma{:}@_a\varphi}{\tau{:}a, \tau{:}\varphi}\,(@)^1 \qquad \frac{\sigma{:}\neg a}{\tau{:}a}\,(\neg)^1$$

$$\frac{\sigma{:}\varphi, \sigma{:}a, \tau{:}a}{\tau{:}\varphi}\,(\nu Id)^2 \qquad \frac{\sigma{:}b, \sigma{:}a, \tau{:}a}{\tau{:}b}\,(nom)$$

[1] The prefix τ is new on the branch.
[2] τ is the earliest introduced prefix in the branch making a true.

Fig. 1. Rules of the prefixed tableaux method for $\mathcal{H}(@)$

the general algorithm is terminating. Finally, HTab and HyLoRes are actually being developed in coordination, and a generic inference system involving both provers is being designed. The aim is to take advantage of the dual behaviour existing between the resolution and tableaux algorithms: while resolution is usually most efficient for unsatisfiable formulas (i.e., a contradiction can be reported as soon as the empty clause is derived), tableaux methods are better suited to handle satisfiable formulas (i.e., a saturated open branch in the tableaux represents a model for the input formula).

4 A Tableaux Method for Hybrid Logics

The tableaux algorithm implemented in HTab is taken from [4] where a terminating decision procedure for hybrid logics up to $\mathcal{H}(@, A, \Diamond^{-1})$ is introduced (currently, HTab implements only the rules for $\mathcal{H}(@)$).

4.1 Rules

The rules of the prefixed tableaux method for the language $\mathcal{H}(@)$ are given on figure 1.

As can be seen in the figure, the rules handle *prefixed formulas*, which are of the form $\sigma{:}\varphi$, for φ a formula of the hybrid language, and $\sigma \in \mathsf{PREF}$, a countable set of symbols called *prefixes*. The interpretation of a prefixed formula $\sigma{:}\varphi$ is that φ is true in a world designated by σ. In addition to prefixed formulas, we notice that the rule $\Diamond$ produces *accessibility formulas*, of the form $\sigma{:}\Diamond\tau$, where σ and τ are prefixes. Such formulas do not belong to the object language, but help the course of the procedure[4].

A tableau for an input formula φ in this calculus is a well-founded, finitely branching tree with root $\sigma{:}\varphi$, and in which each node is labeled by a prefixed formula, and the edges represent applications of tableau rules in the usual way.

A branch is said to be closed if it contains the formulas $\sigma{:}\varphi$ and $\sigma{:}\neg\varphi$, with $\sigma \in \mathsf{PREF}$ and $\varphi \in \mathsf{FORMS}$.

From a direct examination of the rules, we can already discuss some of the main characteristics behind HTab. For example, to avoid useless repeated applications, five of the eight rules ($\wedge$, $\vee$, $\Diamond$, @, $\neg$) can be constrained so that the premise formula is eliminated from the branch once the rule is applied. For the $\Box$ rule on the other hand, it is necessary to keep the two premise formulas after the application of the rule, because they can be used once again separately in other applications. The $\Diamond$ rule has a side condition requiring the prefix to be new in the branch, and hence we should keep track of already used prefixes.

Finally, given the expressivity of the hybrid language (which provides a limited kind of equality between states), prefixes and nominals form equivalence classes intuitively defined by the relation "refer to the same state as." In the course of the procedure, these equivalence classes are created, enlarged and merged. As we will see in the next section, efficiently handling these operations is crucial for an appropriate performance of the prover. The effect of rule (νId) is that the smallest prefix in a given equivalence class should inherit a copy of all the formulas true at any other prefix in the same class. This rule requires a mapping between nominals and the smallest prefix making it true. The rule (*nom*) can be intuitively interpreted as an instruction to merge equivalence classes. Contrary to (νId), it does not impose a direction on the propagation of information. We will see how these two last rules are implemented in the next section.

4.2 Implementation

We will now introduce the main details concerning the implementation of HTab. As the code is released under the GNU GPL, we want to provide some insight on the main algorithms of the system to invite independent development. We will start by describing the structures used in our implementation, then the algorithm implementing the method.

HTab is being developed in the functional language Haskell [9], using the Glasgow Haskell Compiler [6]. It uses a monad structure to define a global state where the main data structure is a *branch.* A branch contains:

- A set of prefixed atomic formulas, of the form $\sigma{:}n$ or $\sigma{:}\neg n$, where $n \in \mathsf{PROP} \cup \mathsf{NOM}$. These are the atomic formulas which are satisfied in the model corresponding to the branch.

[4] In other words, the tableaux rules deal with two sets of symbols – prefixes and nominals – that refer to states in the model. Intuitively, we can think of prefixes are new nominals which are introduced on demand during the application of the tableaux rules, while any nominal appearing in a node of the tableau should appear also in the input formula. Keeping these two kind of symbols apart is useful for ensuring termination of the algorithm.

- Separate sets of prefixed formulas whose main connector is $\wedge$, $\vee$, $\Diamond$, $\Box$, @, or of the kind $\neg$*nominal*. The type of a formula determines the rules that can be applied to it.
- A list *BoxRuleChart*, used to store the pairs (accessibility formula, $\Box$ formula) which have already been used by the $\Box$ rule
- A counter indicating the last prefix created.

The main algorithm can be specified in two steps. First, during the initialisation step the input formula is put into negative normal form, prefixed with the prefix 0 and stored in one of the lists in the branch structure depending of its main connective. The second step is then started taking as input this initial branch.

Algorithm 1 Tableaux algorithm

```
Require: a branch $\mathcal{B}$
Ensure: SAT or UNSAT

if $\mathcal{B}$ is closed then
  return UNSAT
else
  $\mathcal{L_R} \leftarrow$ possibleRulesApplications($\mathcal{B}$)
  if $\mathcal{L_R}$ empty then
    res $\leftarrow$ SAT
  else
    $\mathcal{R} \leftarrow$ chooseRuleAmong($\mathcal{L_R}$)
    $\mathcal{L_B} \leftarrow$ applyRuleOnBranch($\mathcal{R}$, $\mathcal{B}$)
    repeat
      $\mathcal{B}' \leftarrow$ chooseBranchAmong($\mathcal{L_B}$)
      $\mathcal{L_B} \leftarrow \mathcal{L_B}$ - $\mathcal{B}'$
      res $\leftarrow$ apply the algorithm on $\mathcal{B}'$
    until res = SAT or $\mathcal{L_B}$ is empty
  end if
  return res
end if
```

Some of the functions mentioned in Algorithm 1 deserve further comments:

possibleRulesApplications: creates a list of pairs (rule, [formula]) of possible rules applications. To do so, each of the set of formulas of the branch is scanned, with some constraints begin checked (like the one of the rule $\Box$ with *BoxRuleChart*).

applyRuleOnBranch: this function creates one or several branches. Each new branch is created from the current branch, with modifications among the following:

- suppression of a formula (typically, the premise formula),
- addition of one or several formulas (typically, the conclusions of a rule),
- addition of an accessibility formula,
- incrementation of the counter of the last prefix generated in the branch (in the case of the rule $\Diamond$),

- addition of a pair (accessibility formula, $\Box$ rule) in *BoxRuleChart*.

Clash detection consists of detecting $\sigma{:}n$ and $\sigma{:}\neg n$ in the same branch, with $\sigma \in \mathsf{PREF}$ and $p \in \mathsf{PROP} \cup \mathsf{NOM}$. To do so, each prefixed atomic formula added in the branch is saved in a dedicated structure. When this is done, the possibility of a clash is checked. If a clash is detected, the algorithm stops, returning the branch and the culprit formula.

4.2.1 Structures and Invariants for (νId*) and (*nom*)*

To implement the rules (ν*Id*) and (*nom*) we proceed differently than for the other rules. We include these rules in the algorithm as a set of invariants that we ensure every time that a formula is added to the current branch. Thus, the question of saturation is irrelevant in these two cases.

To specify these invariants, let $\mathcal{B}$ be the set of formulas in the current branch, let $\leq$ be an arbitrary order over PREF, let $\mathcal{H}_1 : \mathsf{NOM} \rightarrow \mathsf{PREF}$ be a mapping assigning prefixes to nominals, let $\mathcal{H}_2 : \mathsf{PREF} \rightarrow 2^{\mathsf{FORMS}}$ be a mapping assigning sets of formulas to prefixes, and let $\mathcal{E} : (\mathsf{PREF} \times \mathsf{NOM}) \rightarrow \{\mathit{True}, \mathit{False}\}$ be a Boolean matrix.

- $\mathcal{I}_{\text{min}} : \mathcal{H}_1(a) = \sigma \Leftrightarrow (\sigma{:}a \in \mathcal{B}) \wedge \forall \sigma'.(\sigma'{:}a \in \mathcal{B} \Rightarrow \sigma \leq \sigma')$. This invariant simply characterises $\mathcal{H}_1$ as the function mapping each nominal to the smallest prefix in the branch making the nominal true.
- $\mathcal{I}_{\text{saturation}} : \mathcal{H}_1(a) = \sigma \Leftrightarrow \forall \sigma'.((\sigma'{:}\varphi \in B) \wedge (\sigma'{:}a \in \mathcal{B}) \Rightarrow \sigma{:}\varphi \in \mathcal{B})$. This invariant expresses the necessity that the smallest prefix of a class must retrieve a copy of all the formulas of the other prefixes of the class.
- $\mathcal{I}_{\text{member}} : \varphi \in \mathcal{H}_2(\sigma) \Leftrightarrow \sigma{:}\varphi \in \mathcal{B}$. This invariant characterises $\mathcal{H}_2$ as the function mapping each prefix to the set of formulas that holds in that prefix.
- $\mathcal{I}_{\text{eq}} : \sigma{:}a \in \mathcal{B} \Rightarrow \mathcal{E}_{\sigma,a} = \mathit{True}$. The matrix records the equivalent classes determined by the appearance of formulas of the form $\sigma{:}a$, where a is a nominal, in the branch.
- $\mathcal{I}_{\text{nom}} : \mathcal{E}_{\sigma,b} = \mathcal{E}_{\sigma,a} = \mathcal{E}_{\tau,a} = \mathit{True} \Rightarrow \mathcal{E}_{\tau,b} = \mathit{True}$. This invariant is the direct translation of the rule (*nom*).

Notice that given the order $\leq$ on PREF, the matrix $\mathcal{E}$ enables us to retrieve the minimal prefix for a given equivalence class.

These invariants are equivalent to the use of the rules (ν*Id*) and (*nom*) in a standard tableaux method. The effect of having the rule (ν*Id*) applied with the highest priority among all rules is taken care of by the invariants $\mathcal{I}_{\text{min}}$ and $\mathcal{I}_{\text{saturation}}$. The case is similar for the rule (*nom*) and the invariants $\mathcal{I}_{\text{eq}}$ and $\mathcal{I}_{\text{nom}}$. The invariant $\mathcal{I}_{\text{member}}$ simply prepares the ground for all the other invariants.

Let us now describe how this set of invariants is maintained in HTab.

4.2.2 Maintaining the Invariants

When a formula is added to a branch, two different cases must be handled to maintain the invariants mentioned in the previous section. The simplest case is when a formula $\sigma{:}\varphi$, with $\varphi \notin \mathsf{NOM}$, is added to a branch (see algorithm 2). In this case we only need to ensure that the formula φ is copied to the smallest prefix –

the *urfather* – of the equivalence class. This is because such formulas do not change the equivalent classes defined over $\mathsf{NOM} \cup \mathsf{PREF}$.

Algorithm 2 Maintaining of the invariants when $\sigma{:}\varphi$ ($\varphi \notin \mathsf{NOM}$) is added to the branch

```
$\mathcal{H}_2(\sigma) \leftarrow \mathcal{H}_2(\sigma) \cup \{\varphi\}$        // to maintain $\mathcal{I}_{member}$
$\mathcal{B} \leftarrow \mathcal{B} \cup \{urfather(\sigma){:}\varphi\}$
```

urfather : Prefix → Prefix is the function that, for a given prefix, returns the smallest prefix of its equivalence class (see algorithm 3).

Algorithm 3 *urfather* function

```
Require: $\sigma$ the prefix whom we look for the urfather
Ensure: $\tau$ the smallest prefix in the equivalence class of $\sigma$

  $i_{min} \leftarrow \min\ \{i \mid \mathcal{E}_{\sigma,i} = True\}$
  $\tau \leftarrow \min\ \{j \mid \mathcal{E}_{j,i_{min}} = True\}$
```

The second case, when a formula $\sigma{:}a$, with $a \in \mathsf{NOM}$, is added to the branch, is more complicated. The algorithm 4, handles both sub-cases: when it provokes a merge of two equivalence classes and when it does not. We can sum up this algorithm in two lines:

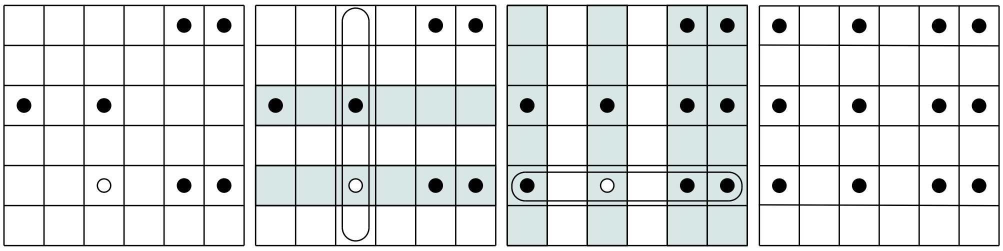
(i) add $\sigma{:}a$ to the equivalence classes, and merge if needed (lines 1 to 5)

(ii) copy formulas of each "old" urfather to the "new" urfather (lines 6 to 16)

An example of the first part is given in figure 2.

Algorithm 4 Maintaining the invariants when $\sigma{:}a$ ($a \in \mathsf{NOM}$) is added to the branch

```
 1: $\mathcal{E}_{\sigma,a} \leftarrow True$                                   // to maintain $\mathcal{I}_{eq}$
 2: $\mathcal{L} \leftarrow \{L_n \mid \mathcal{E}_{\sigma,n} = True\}$
 3: $\mathcal{E} \leftarrow \mathcal{E}$ with the rows of $\mathcal{L}$ replaced by or($\mathcal{L}$)
 4: $\mathcal{C} \leftarrow \{C_\tau \mid \mathcal{E}_{\tau,a} = True\}$
 5: $\mathcal{E} \leftarrow \mathcal{E}$ with the columns of $\mathcal{C}$ replaced by or($\mathcal{C}$)    // to maintain $\mathcal{I}_{nom}$
 6: $iC \leftarrow$ list of the index of the columns of $\mathcal{C}$ ,
 7: oldUrfathers $\leftarrow \{\mathcal{H}_1(n) \mid n \in iC\} \cup \{\sigma\}$
 8: newUrfather $\leftarrow$ min(oldUrfathers)
 9: for $\sigma' \in$ (oldUrfathers − newUrfather) do
10:   for $\varphi \in \mathcal{H}_2(\sigma')$ do
11:     $\mathcal{B} \leftarrow \mathcal{B} \cup \{$newUrfather:$\varphi\}$           // to maintain $\mathcal{I}_{saturation}$
12:   end for
13: end for
14: for $n \in iC$ do
15:   $\mathcal{H}_1(n) \leftarrow$ newUrfather                           // to maintain $\mathcal{I}_{min}$
16: end for
```



(a) Two equivalence classes and a new point (b) ORing these rows (c) ORing these columns (d) Result: one equivalence class

Fig. 2. Example of update of the matrix $\mathcal{E}$

4.3 Optimisations

HTab includes a few optimisations, which are semantic branching, full clash detection and backjumping. They are briefly described below.

Semantic branching: Semantic branching [8] addresses one of the problems of the tableaux method, which is that the different branches of the tree might "overlap" (in terms of the possible models they represent). This leads to superposition of the search space explored by each branch.

The solution consists in adding to the second explored branch the negation of the formula added in the first branch – which is closed. The disjunction rule is replaced by:

$$\frac{\sigma{:}(\varphi \vee \psi)}{\sigma{:}\varphi \mid \sigma{:}(\neg\varphi) \wedge \psi}\ (\textit{semantic branching})$$

Full clash: We can extend clash detection to complete formulas in the hope if detecting clashes earlier in the branch. To do so, formulas should not be transformed into negative normal form. Then, a simple generalisation of the clash-detection structure seen in section 4.2 is all that is required.

The testing we carried out showed that from these two optimisations, semantic branching is the one with the highest impact. While full clash detection results in some improvements, it doesn't seem to be crucial for the system.

Backjumping: Backjumping is an optimisation that aims to reduce search space by replacing systematic one-level-up backtracking by a dependency-directed backtracking. A simple example from [8] is this formula:

$$(A_1 \vee B_1, A_2 \vee B_2, ...A_n \vee B_n) \wedge (\Diamond(A \wedge B)) \wedge (\Box\neg A)$$

Without backjumping we have to explore the whole search space created by the disjunctions in the left, while the causes of the clash – $\Diamond(A \wedge B)$ and $\wedge(\Box\neg A)$ – do not depend on them.

To be able to determine exactly up to which branching point we can backtrack, backjumping requires new information to be attached to each prefixed formula. We decorate each prefixed formula with its "dependency point" which is the branching point (i.e., the particular application of the $\vee$ rule) in which the formula was generated. This information is then propagated to formulas obtained by the application of other rules: a formula depends on a particular branching point if it has been added to the branch at the moment of this particular application of the $\vee$ rule, or if it has been added by the application of a rule where one of the premise formulas depends on this branching.

In order to keep backjumping information, each prefixed formula is decorated by a dependency set and the rules have to be adapted to propagate these dependencies, especially those that have several premise formulas like the ($\Box$) rule:

$$\frac{\sigma{:}d_1{:}\Box\varphi\ ,\ \ \sigma{:}d_2{:}\Diamond\tau}{\tau{:}(d_1{\cup}d_2){:}\varphi}\ (\Box)$$

In addition, we need to also ensure that the invariants that we implemented to account for the (νId) and (nom) rules also propagate dependency information. As the aim of these two rules is to copy formulas from one prefix to another according to the equivalence class they belong to, we choose to keep track of the dependencies of each equivalence class (i.e., the union of the dependencies of all the formulas that have contributed to the class). This is a quite radical solution, as it is not necessary to add the whole dependency set of a class to a copied formula to have a correct implementation of backjumping. The ideal solution would be to strictly keep track of the "path" that links two equivalent prefixes, instead of all contributions to the equivalence class, but the coarser solution we discuss below requires much less book keeping.

The dependencies of an equivalence class are stored in a mapping from the urfathers to the set of dependencies. Let DEP be the enumerable set of dependencies. In our implementation, it is the depth at which a branching occurs. Let $\mathcal{H}_3 : \mathsf{PREF} \to 2^{\mathsf{DEP}}$ be a mapping from prefixes to a set of dependencies. $\mathcal{H}_3$ must meet this invariant:

- $\mathcal{I}_{\text{deps}}$ $(\sigma{:}d{:}n \in \mathcal{B} \wedge \mathcal{H}_1(n) = \sigma) \Rightarrow d \in \mathcal{H}_3(\sigma)$

That is: if a prefixed atomic nominal formula is in the branch, then the dependencies of this formula must be included in the dependencies of the earliest prefix making this nominal true.

Some simple modifications to the algorithm we discussed in section 4.2.2 are sufficient to maintain this new invariant. In order to handle the case when a formula $\sigma{:}d{:}\varphi$, with $\varphi \notin \mathsf{NOM}$, is added to a branch, we replace algorithm 2 by algorithm 5. Notice that the type of $\mathcal{H}_2$ is now $\mathsf{PREF} \to 2^{\mathsf{DEP}\times\mathsf{FORMS}}$, in order to keep track of the dependencies associated to each formula.

Algorithm 5 Propagating dependencies when $\sigma{:}d{:}\varphi$ ($\varphi \notin \mathsf{NOM}$) is added to the branch

```
$\mathcal{H}_2(\sigma) \leftarrow \mathcal{H}_2(\sigma) \cup \{(d, \varphi)\}$
$u \leftarrow \mathit{urfather}(\sigma)$
$d_2 \leftarrow \mathcal{H}_3(u) \cup d$
$\mathcal{B} \leftarrow \mathcal{B} \cup \{(u{:}d_2{:}\varphi)\}$
```

For the second case, when a formula $\sigma{:}a$ with $a \in \mathsf{NOM}$ is added to the branch, we do the following two additions to the algorithm 4. First, we have to calculate the dependencies of the resulting merge of classes, which is the union of the dependencies of the old classes, together with the dependencies of the formula that triggers the merge (the code is given in algorithm 6, and it should be added just after line 8 in algorithm 4). Second, we still have to copy all the formulas of the old class to the new class, without forgetting to add the dependencies (i.e., we should replace lines 9 to 13 in the previous algorithm with the lines shown in algorithm 7).

Algorithm 6 Maintaining $\mathcal{I}_{\text{deps}}$ when $\sigma{:}d{:}a$ ($a \in \mathsf{NOM}$) is added to the branch

```
newDeps $\leftarrow d$
for $o \in$ oldUrfathers do
    newDeps $\leftarrow$ newDeps $\cup\, \mathcal{H}_3(o)$
end for
$\mathcal{H}_3$(newUrfather) $\leftarrow$ newDeps
```

Algorithm 7 Propagating dependencies when $\sigma{:}d{:}a$ ($a \in \mathsf{NOM}$) is added to the branch

```
for $\sigma' \in$ (oldUrfathers $-$ newUrfather) do
    for $(d, \varphi) \in \mathcal{H}_2(\sigma')$ do
        $\mathcal{B} \leftarrow \mathcal{B} \cup \{$newUrfather:($d \cup$newDeps):$\varphi\}$
    end for
end for
```

The effect of backjumping on performance can be seen in figure 4, where HTab 1.0 and HTab 1.1, HTab without and with backjumping respectively, are compared.

5 Tests

To evaluate the performance of HTab, we use a suite of test scripts originally developed for HyLoRes. The tests are launched on batches of random hybrid formulas. They are done by steps of bigger and bigger formula sizes.

We will compare the performance of HTab with both HyLoRes and HyLoTab on formulas of $\mathcal{H}(@)$ that contain 2 propositional symbols, 2 nominals, 1 relational symbol, and a modal depth of 2. We go from formulas of size 1 to formulas of size 66, in number of conjunctions of clauses. The percentage of satisfiability of the input formulas can be seen on figure 3 (as reported by HyLoRes, the system with the smallest number of timeouts): we go from mostly satisfiable formulas to mostly unsatisfiable ones. As it is in general the case, timeouts occur mostly in the area of maximum uncertainty, where the percentage of satisfiable and unsatisfiable formulas is roughly the same.

We can see the results on figure 4. HyLoTab is far behind the two other provers. Concerning HTab and HyLoRes, we see that their curves cross in the point corresponding to 22 clauses. Before this point, HTab behaves better than both HyLoTab and HyLoRes. This is because the tableaux method generally terminates faster than resolution when a formula is satisfiable. After the 22 clause point, HyLoRes starts to gain the upper hand. HTab 1.1 is still much slower than HyLoRes on these formulas, but thanks to backjumping, it remains farily well behaved (as we can see HTab 1.0 would mostly timeout in all this area). HTab 1.1 median times go down after about 50 clauses as a combined result of backjumping and semantic branching.

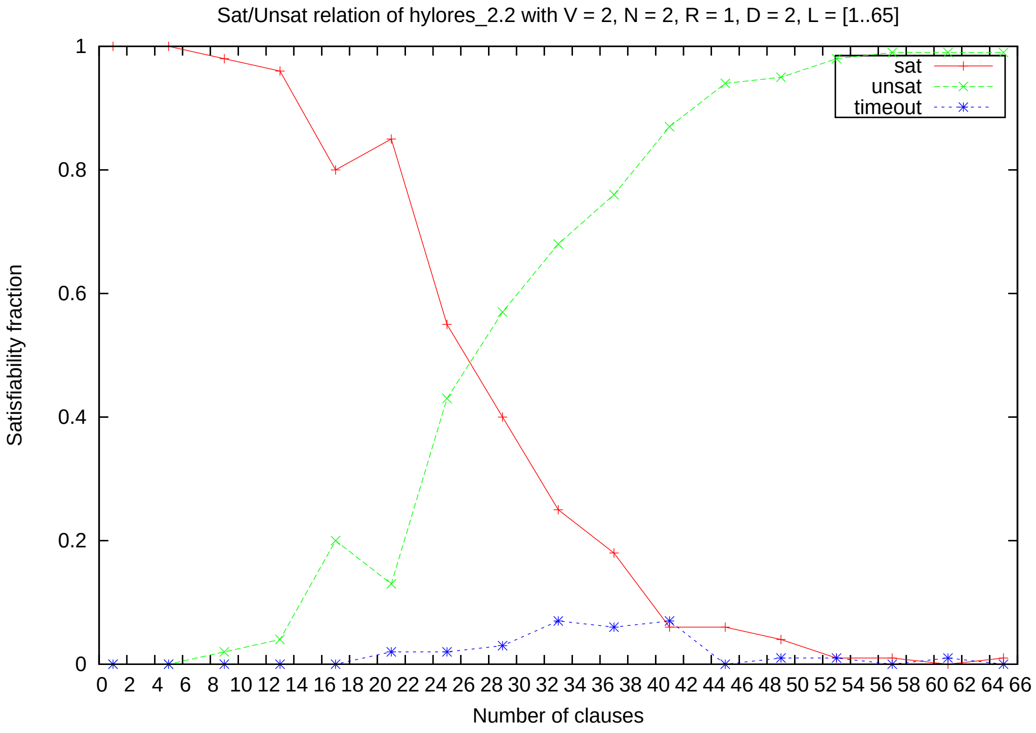


Fig. 3. SAT/UNSAT/Timeout repartition of the formulas for HyLoRes

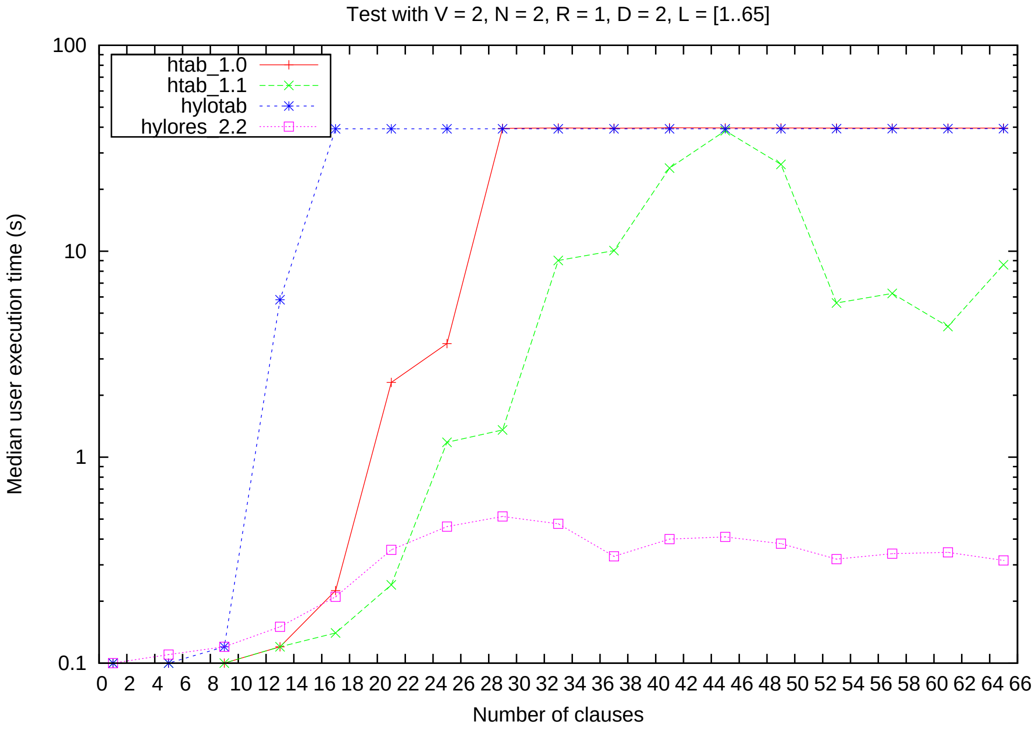


Fig. 4. Median time of execution between HyLoTab, HTab (versions 1.0 and 1.1) and HyLoRes

6 Example of Use

As an input, HTab takes a file containing a set of formulas. The syntax used can be seen with this sample input file:

```
begin
<>[](p1 v p2) & []<>(p2 v p1) & <><>(p1) & <><>(p2);
([](-p1 v -p2) & [](p3 <->p1)) & ([](p1 <-> p2 & [](p2 <-> p1)));
(@ n1 (p1 <-> p3) & (@ n2 (p1 <-> p2)) & (@ n1 -n2))
end
```

Executing HTab on these formulas is done with this call:

```
$ htab -f test.frm
Reading parameters from .htabrc

The formula is satisfiable.
(final statistics)
begin
----------------------------------
Closed branches: 68
----------------------------------
end
Elapsed time: 3.6002e-2
```

The argument `-gm filename` can be added in order to generate a model and write it into the file `filename`. The model found for the previous formula is:

```
Model{
 worlds = fromList [N0,N1,N2,N3,N4,N5,N6,N7,N8,N9,N10],
 succs  = [(N2,R1,N3),(N2,R1,N5),(N2,R1,N7),(N3,R1,N4),
           (N3,R1,N10),(N5,R1,N6),(N5,R1,N9),(N7,R1,N8)],
 valP   = [(P1,fromList [N0,N1,N6,N8,N9,N10]),
           (P2,fromList [N1,N4]),
           (P3,fromList [N0])],
 valN   = [(N0,N0),(N1,N1),(N2,N2),(N3,N3),(N4,N4),(N5,N5),(N6,N6),
           (N7,N7),(N8,N8),(N9,N9),(N10,N10),(N11,N0),(N12,N1)],
 sig    = Sig {nomSymbols = fromList [N0,N1,N2,N3,N4,N5,N6,
                                      N7,N8,N9,N10,N11,N12],
               propSymbols = fromList [P1,P2,P3],
               relSymbols = fromList [R1]}}
```

7 Conclusion

We have implemented a preliminary version of a prover for hybrid logic based on tableaux method, guaranteeing termination for all input formulas of $\mathcal{H}(@)$.

Although we are still at an early stage of implementation, the performance we get is encouraging. There is still plenty of room for optimisations on both the internal data structures used and on the tableau algorithm itself. For example, although the algorithm we are using to update the equivalence classes of prefixes and nominals has been optimised, its implementation uses copies of structures. As the algorithm in itself is already fairly complex, we have decided to first implement a correct version with unoptimised data types that require copying of big structures, and this seriously slow down the prover. We can explore two ways to solve this problem. The first one is to improve this matrix system by using dynamic memory allocation, and also examining the number of duplications of formulas caused by the (νId) rule in order to find ways to reduce them. The other solution is, as suggested in [10], to use a disjoint-set forest to represent equivalence classes instead.

We have not yet implemented some optimisations of the basic tableaux algorithm which are standards in state-of-the-art tableaux-based provers like racer. (e.g., model caching).

Once the basic hybrid logic is tamed, our next goal is to implement frame conditions, like reflexivity or transitivity, by using the current work of Bolander and Blackburn (see [3]).

References

[1] C. Areces, P. Blackburn, and M. Marx. A road-map on complexity for hybrid logics. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic*, number 1683 in LNCS, pages 307–321. Springer, 1999. Proceedings of the 8th Annual Conference of the EACSL, Madrid, September 1999.

[2] C. Areces and D. Gorín. Ordered resolution with selection for H(@). In F. Baader and A. Voronkov, editors, *Proceedings of LPAR 2004*, volume 3452 of *LNCS*, pages 125–141. Springer, 2005.

[3] T. Bolander and P. Blackburn. Decidable tableau calculi for modal and temporal hybrid logics extending **K**, 2007. Method for Modalities 5, 2007.

[4] T. Bolander and P. Blackburn. Termination for hybrid tableaus. *Journal of Logic and Computation*, 17:517–554, 2007.

[5] J. van Eijck. HyLoTab — Tableau-based theorem proving for hybrid logics. manuscript, CWI, available from http://www.cwi.nl/~jve/hylotab, 2002.

[6] GHC, The Glasgow Haskell Compiler. http://www.haskell.org/ghc/. Last visited: 15/09/07.

[7] V. Haarslev and R. Möller. RACER system description. *Lecture Notes in Computer Science*, 2083:701–705, 2001.

[8] I. Horrocks and P. Patel-Schneider. Optimising description logic subsumption, 1998.

[9] S. Peyton Jones and J. Hughes (editors). Haskell 98: A non-strict, purely functional language. Technical report, Haskell.org, 1999.

[10] Mark Kaminski and Gert Smolka. A straightforward saturation-based decision procedure for hybrid logic. In Jørgen Villadsen, Thomas Bolander, and Torben Braüner, editors, *International Workshop on Hybrid Logic 2007 (HyLo 2007)*, Dublin, Ireland, 2007.