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 hybrid logic $\mathcal{H}(@, \mathsf{A})$. The performance of the tableaux algorithm is compared with the performances of HyLoRes, HyLoTab (a system based on a different tableaux algorithm) and RacerPro.
HTab is written 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 hybrid logic $\mathcal{H}(@, \mathsf{A})$ described by Bolander and Blackburn 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 [12]), and the description logic prover RacerPro (see [7]). 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 $\mathcal{H}(@, \mathsf{A})$. In Section 3, we

[1] Email: guillaume.hoffmann@loria.fr
[2] Email: carlos.areces@loria.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

discuss 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 optimisations. In Section 5, we list the result of testing. In the conclusion we see the perspectives for further developments of the prover.

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

$\mathcal{H}(@, \mathsf{A})$ extends the basic modal language by adding nominals, the satisfaction operator and the universal modality. 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 $\mathcal{H}(@, \mathsf{A})$ in the signature $\langle \mathsf{REL}, \mathsf{PROP}, \mathsf{NOM} \rangle$ are given by the following recursive definition:

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

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 $V : \mathsf{PROP} \cup \mathsf{NOM} \to \wp(M)$ 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 the elements of M *states*, *worlds* or *points*. 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:

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

We will use the following standard notations: $\varphi_1 \vee \varphi_2$ for $\neg((\neg\varphi_1) \wedge (\neg\varphi_2))$, $\Box\varphi$ for $\neg\Diamond\neg\varphi$, and $\mathsf{E}\varphi$ for $\neg\mathsf{A}\neg\varphi$.

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. However, this problem is EXPTIME-complete for $\mathcal{H}(@, \mathsf{A})$ (see [11] and [8]).

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.) for which termination can be ensured. 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 1.3.2 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 only handle the hybrid logic $\mathcal{H}(@, \mathsf{A})$ over the class of all frames.

Even though other provers for languages similar to $\mathcal{H}(@, \mathsf{A})$ exist, 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.

- RacerPro [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.). However, the language $\mathcal{ALCQHI}_{\mathcal{R}^+}$ is incomparable with $\mathcal{H}(@, \mathsf{A})$. Intuitively, it has a restricted use of @, and nominals are not allowed.
- HyLoTab [12] is a tableaux based prover for the hybrid logics up to $\mathcal{H}(@, \Diamond^{-1}, \downarrow, \mathsf{A})$ ($\Diamond^{-1}$ is the inverse modality and $\downarrow$ is the 'bind-to-the-current-state' binder). 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 subfragments of $\mathcal{H}(@, \Diamond^{-1}, \downarrow, \mathsf{A})$ like $\mathcal{H}(@, \mathsf{A})$.[3]
- HyLoRes [2] is a resolution based prover for the hybrid logics up to $\mathcal{H}(@, \Diamond^{-1}, \downarrow, \mathsf{A})$. The implemented algorithm is terminating for formulas in $\mathcal{H}(@, \Diamond^{-1})$ — but not $\mathcal{H}(@, \mathsf{A})$ — 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 when run on decidable fragments of $\mathcal{H}(@, \Diamond^{-1}, \downarrow, \mathsf{A})$ is EXPTIME.

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

[3] For instance, the formula $\mathsf{A}\Diamond p$ makes HyLoTab loop.

features distinguishes it from RacerPro. On the negative side, the current version of HTab has only a few optimisations, while RacerPro is a mature theorem prover that includes most state-of-the-art optimisation techniques. HyLoTab is the system most similar to HTab, both being tableaux based provers for hybrid logic. Besides some technical issues (the way HyLoTab handles nominals equality differs from the approach taken in HTab) the main difference is termination: one of the main aims of HTab is to always ensure that the general algorithm is terminating, which HyLoTab does not. 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 are adapted from [4] where a terminating decision procedure for hybrid logics up to $\mathcal{H}(@, \mathsf{A}, \Diamond^{-1})$ is introduced. Currently, HTab handles only the hybrid logics up to $\mathcal{H}(@, \mathsf{A})$.

4.1 Rules

The rules of the prefixed tableaux method for the language $\mathcal{H}(@, \mathsf{A})$ are given on Figure 1. [4]

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. [5]

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. At the beginning of the calculus, the tableau must also contain one node of the shape $\sigma(n){:}n$ per nominal n of the input formula, with $\sigma(n)$ being a fresh prefix.

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,

[4] Instead of the non-directed (Id) rule used for $\mathcal{H}(@, \mathsf{A})$ in [4], we keep using the (νId) rule of the calculus for $\mathcal{H}(@)$, which still ensures correctness, completeness and termination.

[5] 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 as 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.

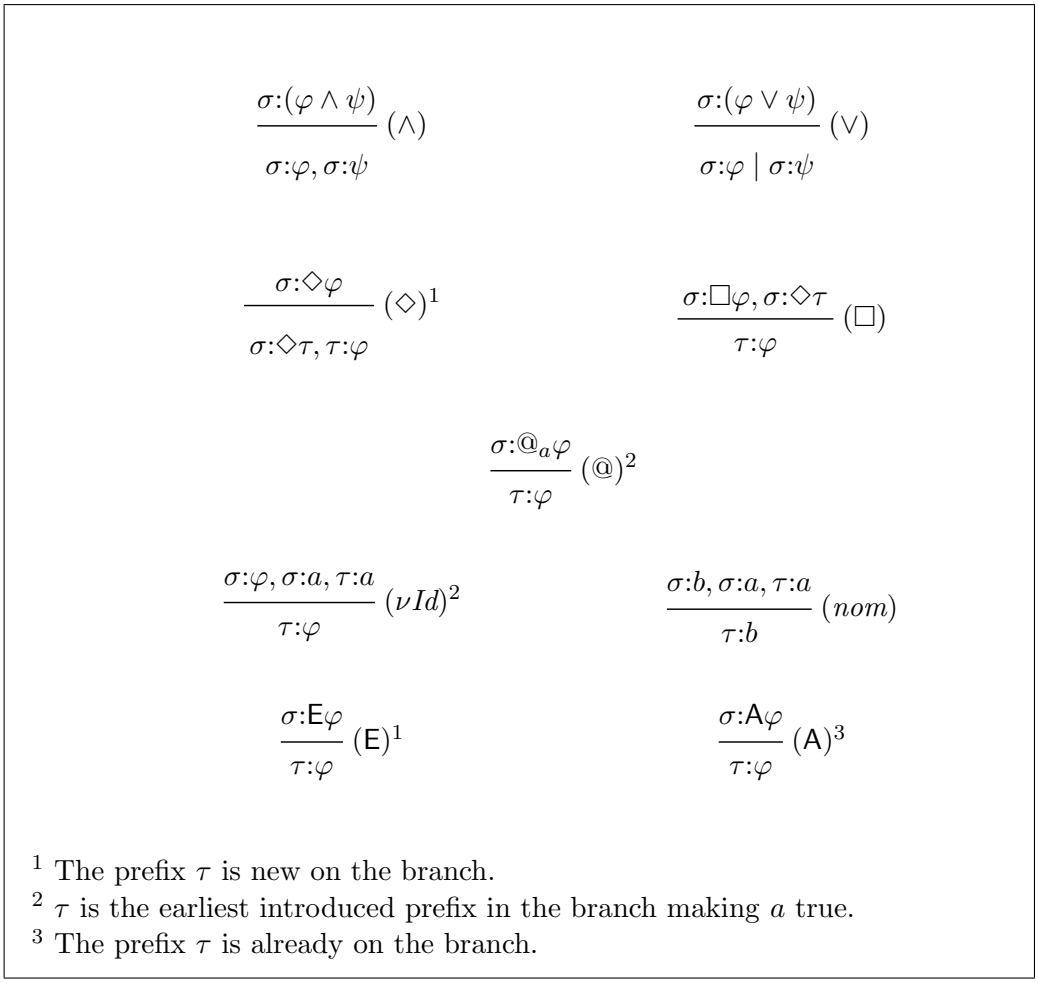

Fig. 1. Rules of the prefixed tableaux method for $\mathcal{H}(@, \mathsf{A})$

five of the nine rules — $(\wedge)$, $(\vee)$, $(\Diamond)$, $(@)$, (E) — can be constrained so that the premise formula is eliminated from the branch once the rule is applied. For the (A) rule, we put the premise formula without its prefix in a set of constraint formulas that are added to all prefixes of the branch. Similarly, for the $(\Box)$ rule, we associate each prefix with a set of $\Box$-constraints, that send formulas to accessible prefixes.

The rules $(\Diamond)$ and (E) are called *prefix generating rules*. Their saturation condition is that they can not be applied twice on the same prefixed formula on the same branch. Given that the (E) rule is a global requirement on the model, we can prevent its double application on the same non-prefixed formula.

Finally, given the expressiveness 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. The effect of the 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. The rule (nom) can be interpreted as an instruction to merge equivalence classes. 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 a copyleft license and managed with a distributed version control system, 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, then the algorithm implementing the method.

HTab is being developed in the functional language Haskell [10], 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:

- Clashable formulas: a map associating to a prefix a set of atomic formulas, of the form n or $\neg n$, where $n \in \mathsf{PROP} \cup \mathsf{NOM}$. These are the atomic formulas which are satisfied in the model corresponding to the branch.
- Pending formulas: separate sets of prefixed formulas whose main connector is $\wedge$, $\vee$, $\Diamond$, @ or E. The type of a formula determines the rule that can be applied to it.
- Accessibility relations: for a given prefix and a given relation, which prefixes are accessible. This structure may be augmented when the ($\Diamond$) rule is applied.
- $\Box$-constraints: for a given prefix and a given relation, which formulas are forced to be true at the accessible prefixes. This structure may be augmented when the ($\Box$) rule is applied.
- Universal constraints: a set of formulas that have to be true at all prefixes of the branch. This structure may be augmented when the (A) rule is applied.
- Some charts to handle rules that require book keeping:
  - for the ($\Diamond$) rule, a chart that checks that this rule is never applied twice on the same prefixed input formula.
  - for the (E) and (@) rules respectively, a chart that checks that theses rules are never applied twice on the same non-prefixed input formula.
- A counter indicating the last prefix created.
- An inclusion forest of the sets of formulas that hold at each prefix.

We can see from this description that we have two kind of rules:

- immediate rules — ($\Box$) and (A) — that are fired as soon as a formula of the corresponding type is added to the branch, and as soon as a new prefix can be constrained by these rules. [6]
- delayed rules — ($\wedge$), ($\vee$), ($\Diamond$), (@) and (E) — that are fired on pending formulas, and thus whose application order can be specified.

The rules (νId) and (nom) can also be seen as immediate, but as we will see in the next section, their implementation is not comparable to the others.

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

[6] Making these rules immediate eliminates the need for possibly resource-intensive book keeping. Indeed, previous versions of HTab that had a delayed application of the ($\Box$) rule would spend up to 20% of the running time by checking its saturation condition.

0 and added to the branch. Also, for each nominal present in the input formula, a prefixed atomic formula with a fresh prefix is added to the branch, to comply with the semantics of nominals. The second step, a traditional backtracking tableaux algorithm, is then started taking as input this initial branch.

Clash detection consists in spotting $\sigma{:}n$ 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 to the branch is saved in the "clashable formulas" map, after checking that it does not cause a clash. If a clash is detected, the algorithm stops, returning the branch and the culprit formula.

4.2.1 Structures and Invariants

We are now going to see how we implement the rules (νId), (nom), and the blocking condition required to have a terminating calculus for $\mathcal{H}(@, \mathsf{A})$. We capture the behaviour of the rules (νId) and (nom) by adding new structures to those shown in the previous section, and ensuring a set of invariants on them every time a formula is added to the current branch.

Let $\mathcal{B}$ be the set of formulas in the current branch. Let $\mathcal{H}_1 : \mathsf{PREF} \to 2^{\mathsf{FORMS}}$ be a mapping assigning sets of formulas to prefixes. Let $\leq_1$ be a well-order over PREF and let $\leq_2$ be a well-order over NOM. Thus, we can define a well-order $\leq$ on $\mathsf{PREF} \cup \mathsf{NOM}$ by:

- $\forall \sigma_1, \sigma_2 \in \mathsf{PREF}$, $\sigma_1 \leq \sigma_2$ iff $\sigma_1 \leq_1 \sigma_2$
- $\forall n_1, n_2 \in \mathsf{NOM}$, $n_1 \leq n_2$ iff $n_1 \leq_2 n_2$
- $\forall \sigma \in \mathsf{PREF}$ and $\forall n \in \mathsf{NOM}$, $\sigma \leq n$

Let $\mathcal{F}$ be a disjoint-set forest over elements of $\mathsf{PREF} \cup \mathsf{NOM}$, as described in [5]. $\mathcal{F}$ is equipped with the usual functions *find* — that returns the representative of a set — and *union* — that merges two sets.

We maintain the following invariant on $\mathcal{F}$:

- $\mathcal{I}_{\text{representative}}$: a representative in $\mathcal{F}$ is the $\leq$-smallest element of its set.

As in HTab we may modify equivalence classes only when adding formulas of the form $\sigma{:}a$ with $\sigma \in \mathsf{PREF}$ and $a \in \mathsf{NOM}$ to the branch, we work without the usual *makeSet* function used to create a singleton set, and instead use the *union* function. Therefore, the only function that modifies the disjoint-set forest is *union*. When performing this function, we maintain $\mathcal{I}_{\text{representative}}$ by setting as representative of the resulting set the $\leq$-smallest representative of the sets which the inputs belong to. If an input does not belong to a set of $\mathcal{F}$ yet, we do as if it belonged to a singleton set containing itself.

This invariant has a useful consequence: as we always add couples made of a nominal and a prefix to the equivalence classes, no class can exist without containing a prefix. Then, by definition of $\leq$, the smallest element has to be a prefix. So we know that the representative of an equivalence class is always a prefix. We will also qualify as "representatives", prefixes that belong to no equivalence class.

We can now specify the invariants that we want these structures — $\mathcal{B}$, $\mathcal{F}$ and $\mathcal{H}_1$ — to verify :

- $\mathcal{I}_{\text{saturation}} : \text{find}(\mathcal{F}, a) = \sigma \Leftrightarrow \forall\sigma'.((\sigma'{:}\varphi \in \mathcal{B}) \wedge (\sigma'{:}a \in \mathcal{B}) \Rightarrow \sigma{:}\varphi \in \mathcal{B})$. This invariant expresses the constraint that the representative a class must retrieve a copy of all the formulas of the other prefixes of the class.
- $\mathcal{I}_{\text{eq}} : \sigma{:}a \in \mathcal{B} \Rightarrow \text{find}(\mathcal{F}, \sigma) = \text{find}(\mathcal{F}, a)$. The disjoint-set structure records the equivalent classes determined by the appearance of formulas of the form $\sigma{:}a$, with $\sigma \in \mathsf{PREF}$ and $a \in \mathsf{NOM}$, in the branch.
- $\mathcal{I}_{\text{member}} : \sigma{:}\varphi \in \mathcal{B} \Leftrightarrow \varphi \in \mathcal{H}_1(\sigma)$. This invariant characterises $\mathcal{H}_1$ as the function mapping each prefix to the set of formulas that holds in that prefix.

Maintaining these invariants is equivalent to the use of the rules (νId) and (nom) in a standard tableaux method with the highest priority. The effect of having the rule (νId) applied with the highest priority among all rules is taken care of by the invariant $\mathcal{I}_{\text{saturation}}$. Similarly, the rule (nom) is implemented through the invariant $\mathcal{I}_{\text{eq}}$ and the equivalence classes in $\mathcal{F}$.

The invariant $\mathcal{I}_{\text{member}}$ enables to ensure termination for the calculus with the universal modality. With this invariant, we can implement a blocking condition on prefix generating rules, as described in [4].

We define the *inclusion urfather* of a prefix σ on a branch $\mathcal{B}$, written $u_{\mathcal{B}}(\sigma)$, to be the earliest introduced prefix τ for which $\mathcal{H}_1(\sigma) \subseteq \mathcal{H}_1(\tau)$. A prefix σ is called an *inclusion urfather* on $\mathcal{B}$ if $\sigma = u_{\mathcal{B}}(\tau)$ for some prefix τ. We can now enunciate the blocking condition: *prefix generating rules can only be applied at prefixes that are inclusion urfathers*. Given that among the two prefix generating rules, namely $(\Diamond)$ and (E), the latter is already constrained to be fired once per non-prefixed premise formula, we need to check this blocking condition only for the $(\Diamond)$ rule.

The calculation of the inclusion urfathers on a branch requires the calculation of the inclusion ordering on the sets of formula that hold at the prefixes of the branch. For now, we only have a preliminary implementation of this, but the performance of this operation should be improved in following versions of the prover.

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 1). In this case we only need to ensure that the formula φ is copied to the representative prefix of the equivalence class.

Algorithm 1 Adding a formula $\sigma{:}\varphi$ ($\varphi \notin \mathsf{NOM}$) to the branch

1: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\sigma{:}\varphi\}$
2: $\mathcal{H}_1(\sigma) \leftarrow \mathcal{H}_1(\sigma) \cup \{\varphi\}$ *// to maintain $\mathcal{I}_{member}$*
3: $r \leftarrow \text{find}(\mathcal{F}, \sigma)$
4: $\mathcal{H}_1(r) \leftarrow \mathcal{H}_1(r) \cup \{\varphi\}$ *// to maintain $\mathcal{I}_{member}$*
5: $\mathcal{B} \leftarrow \mathcal{B} \cup \{r{:}\varphi\}$

The second case, when a formula $\sigma{:}a$, with $a \in \mathsf{NOM}$, is added to the branch, is more complicated. Algorithm 2 handles both sub-cases: when it provokes a merge

of two equivalence classes and when it does not. We can highlight two important parts of this algorithm:

(i) merge the classes of σ and a if needed (line 3)

(ii) copy formulas of each "old" representative to the "new" one (lines 4 to 12)

The first part is handled by the *union* function of the disjoint-set forest.

Algorithm 2 Adding a formula $\sigma{:}a$ ($a \in \mathsf{NOM}$) to the branch

1: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\sigma{:}a\}$
2: $\mathcal{H}_1(\sigma) \leftarrow \mathcal{H}_1(\sigma) \cup \{a\}$ // *to maintain* $\mathcal{I}_{member}$
3: $\mathcal{F} \leftarrow \text{union}(\mathcal{F}, \sigma, a)$ // *to maintain* $\mathcal{I}_{eq}$
4: nomRepr $\leftarrow$ if $a \in \mathcal{F}$ then find($\mathcal{F}$,a) else $\emptyset$
5: involvedReprs $\leftarrow$ {find($\mathcal{F}, \sigma$)}$\cup$ nomRepr
6: newRepr $\leftarrow$ min(involvedReprs)
7: **for** $\sigma' \in$ (involvedReprs $-$ newRepr) **do**
8: **for** $\varphi \in \mathcal{H}_1(\sigma')$ **do**
9: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{newRepr}{:}\varphi\}$ // *to maintain* $\mathcal{I}_{saturation}$
10: $\mathcal{H}_1(\text{newRepr}) \leftarrow \mathcal{H}_1(\text{newRepr}) \cup \{\varphi\}$ // *to maintain* $\mathcal{I}_{member}$
11: **end for**
12: **end for**
13: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{newRepr}{:}a\}$
14: $\mathcal{H}_1(\text{newRepr}) \leftarrow \mathcal{H}_1(\text{newRepr}) \cup \{a\}$ // *to maintain* $\mathcal{I}_{member}$

4.3 Optimisations

HTab includes as optimisations semantic branching, full clash detection and back-jumping. They are briefly described below. After that, we will also explain how we reduced the number of duplicate calculations compared to the original calculus.

Semantic branching: Semantic branching [9] 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 semantic branching had a positive impact on performance. On the other hand, full clash made the calculus run slower in many cases. This optimisation is thus disabled by default.

Backjumping: Backjumping is an optimisation that aims to reduce search space by replacing systematic one-level-up backtracking by dependency-directed backtracking. A simple example from [9] 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 on 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 prefixed formulas. We decorate each prefixed formula with its "dependency points" which are the branching point — i.e., the particular applications of the $(\vee)$ rule — because of 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. 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 also need to 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 representatives to the set of dependencies. Let DEP be the enumerable set of dependencies. In our implementation, a dependency is the depth of the branch at which a branching occurs. Let $\mathcal{H}_2 : \mathsf{PREF} \to 2^{\mathsf{DEP}}$ be a mapping from prefixes to a set of dependencies. $\mathcal{H}_2$ must meet this invariant:

- $\mathcal{I}_{\text{deps}}$ $(\sigma{:}d{:}n \in \mathcal{B} \wedge \text{find}(\mathcal{F}, n) = \sigma) \Rightarrow d \in \mathcal{H}_2(\sigma)$

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 algorithms 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 1 by Algorithm 3.

Notice that the type of $\mathcal{H}_1$ is now $\mathsf{PREF} \to 2^{\mathsf{DEP}\times\mathsf{FORMS}}$, in order to keep track of the dependencies associated to each formula.

Algorithm 3 Adding a formula $\sigma{:}d{:}\varphi$ ($\varphi \notin \mathsf{NOM}$) to the branch

1: $\mathcal{B} \leftarrow \mathcal{B} \cup \{(\sigma{:}d{:}\varphi)\}$
2: $\mathcal{H}_1(\sigma) \leftarrow \mathcal{H}_1(\sigma) \cup \{(d, \varphi)\}$
3: $r \leftarrow \text{find}(\mathcal{F}, \sigma)$
4: $d_2 \leftarrow \mathcal{H}_2(r) \cup d$
5: $\mathcal{B} \leftarrow \mathcal{B} \cup \{(r{:}d_2{:}\varphi)\}$
6: $\mathcal{H}_1(r) \leftarrow \mathcal{H}_1(r) \cup \{(d_2, \varphi)\}$

For the second case, when a formula $\sigma{:}a$ with $a \in \mathsf{NOM}$ is added to the branch, we change Algorithm 2 by Algorithm 4 by doing the two following additions: first, we 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; second, we still copy all the formulas of the old class to the new class, without forgetting to add the dependencies.

Algorithm 4 Adding a formula $\sigma{:}d{:}a$ ($a \in \mathsf{NOM}$) to the branch

1: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\sigma{:}d{:}a\}$
2: $\mathcal{H}_1(\sigma) \leftarrow \mathcal{H}_1(\sigma) \cup \{(d, a)\}$ *// to maintain $\mathcal{I}_{member}$*
3: $\mathcal{F} \leftarrow \text{union}(\mathcal{F}, \sigma, a)$ *// to maintain $\mathcal{I}_{eq}$*
4: nomRepr $\leftarrow$ if $a \in \mathcal{F}$ then find($\mathcal{F}$,a) else $\emptyset$
5: involvedReprs $\leftarrow$ {find($\mathcal{F}, \sigma$)}$\cup$ nomRepr
6: newRepr $\leftarrow$ min(involvedReprs)
7: newDeps $\leftarrow d$
8: **for** $r \in$ involvedReprs **do**
9: newDeps $\leftarrow$ newDeps $\cup\, \mathcal{H}_2(r)$
10: **end for**
11: $\mathcal{H}_2$(newRepr) $\leftarrow$ newDeps
12: **for** $\sigma' \in$ (involvedReprs $-$ newRepr) **do**
13: **for** $(d_2, \varphi) \in \mathcal{H}_1(\sigma')$ **do**
14: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{newRepr}{:}(d_2 \cup \text{newDeps}){:}\varphi\}$
15: $\mathcal{H}_1(\text{newRepr}) \leftarrow \mathcal{H}_1(\text{newRepr}) \cup \{(d_2 \cup \text{newDeps}, a)\}$ *// to maintain $\mathcal{I}_{member}$*
16: **end for**
17: **end for**
18: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{newRepr}{:}(d \cup \text{newDeps}){:}a\}$
19: $\mathcal{H}_1(\text{newRepr}) \leftarrow \mathcal{H}_1(\text{newRepr}) \cup \{(d \cup \text{newDeps}, a)\}$ *// to maintain $\mathcal{I}_{member}$*

While using backjumping, we can easily use the following heuristic: when we apply a rule, we always choose the formulas whose earliest branching dependency is the smallest on the branch. The aim is to boost the effect of backjumping. Indeed, tests have shown that, while backjumping alone has a positive impact on performance, the previous heuristic enables HTab to behave an order of magnitude faster than before.

Preventing duplicates: While the calculus described in [4] involves a lot of duplicate formulas, we have managed to eliminate many of them in HTab.

The situation we want to avoid is applying the same rule twice on the same formula that would be at two different prefixes of the same equivalence class. For this, we have done two changes. First, a new prefixed formula in the branch is only added to the representative of the class of its prefix.

The second change consists in stopping copying the formulas that hold at an ex-representative to the new representative prefix after the merge of two equivalence classes. We know we can do this because, since the previous change, applying a rule on a prefixed formula always yields formulas prefixed by the representative of the class. Nevertheless, we still need to copy the formulas that belong to the clashable data of a given prefix. Indeed, they are — or contain, when full clash is enabled — formulas on which no rule can be applied. So, we still copy this information when we merge two equivalence classes, and test for a clash at this moment.

Now that we no longer copy formulas to representative prefixes, we have to loosen the blocking condition of the calculus of $\mathcal{H}(@, \mathsf{A})$ to: *a prefix generating rule can only be applied at a prefix whose representative is an inclusion urfather.*

We also ensure that the $\Box$-constraints and the accessibility relations always originate at a representative prefix, and we combine those who meet at the same prefix after a merge.

We do not need to change the destination prefix of the accessibility relations, again thanks to the first change. Therefore, this and the new expeditions of formulas done when two equivalence classes merge are two moments when duplications can still happen.

From the viewpoint of rules, these changes translate as deleting the (νId) rule, and in all other rules, replacing the prefixes of the conclusion formulas by the representative prefixes of their equivalence class. The saturation conditions of the (E) and $(@)$ rules do not change, as they do not depend on the prefix of the input formula. On the other hand, the saturation data for the $(\Diamond)$ rule is now stored and used modulo the equivalence class of the prefixes.

These changes have been proven to have a positive impact on performance. We still work with a prefixed calculus, but we get closer to the absence of redundancy of the internalized counterpart, without having to scan all the formulas of the branch to rewrite nominals when two of them are found equal.

Using as little as we need: If the input formula has no existential or universal modality, we do not need the structure $\mathcal{H}_1$, and thus can do the economy of any operation associated to it. Indeed, with the previous modifications, $\mathcal{H}_1$ is now only used to calculate the inclusion urfathers of the branch.

5 Tests

To evaluate the performance of HTab, we use a suite of test scripts that launch provers on batches of bigger and bigger random formulas.

First, we have compared the performance of HTab with both HyLoRes and HyLoTab on formulas of $\mathcal{H}(@)$ that contain 2 propositional symbols, 5 nominals, with a modal depth of 2. We go from formulas of size 1 to formulas of size 91, in number of conjunctions of clauses. The percentage of satisfiability of the input formulas can be seen on Figure 2 (as reported by HTab, the system with the smallest number of

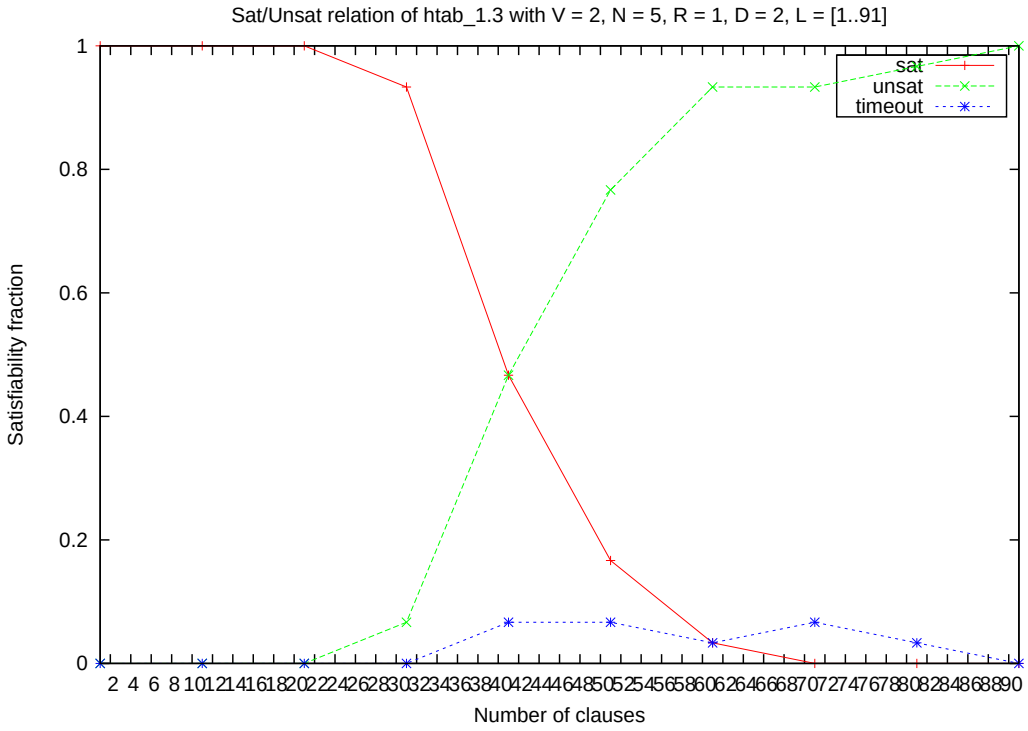

Fig. 2. SAT/UNSAT/Timeout repartition of the formulas for HTab ($\mathcal{H}(@)$)

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 3. HyLoTab is far behind the two other provers. Concerning HTab and HyLoRes, we see that they remain under the limit of 20 seconds of execution, independently of the size of the input formula, but HTab behaves always better than HyLoRes.

To check the implementation of the universal modality, we have compared HTab with RacerPro 1.9.2 beta on formulas of $\mathcal{M}(\mathsf{A})$, the basic modal logic with the universal modality. This time, the formulas contain 3 propositional symbols, and range from size 10 to 70 clauses. We generate formulas with one universal modality clause, followed by basic modal clauses. Then we translate them into $\mathcal{ALC}$ description logic by defining one atomic concept per propositional symbol, and defining the TBox as the content of the universal clause. We define another atomic concept as the translation of the rest of the modal clauses into description logic, and then query the satisfiability of this concept with RacerPro.

The percentage of satisfiability of the input formulas can also be seen on Figure 4, and the results on Figure 5. While HTab is faster than RacerPro on formulas up to 30 clauses, it then becomes much slower, while RacerPro keeps an almost constant median time of execution. This is because as the number of prefixes grows in a branch, calculating the inclusion ordering on their set of formulas becomes more and more costly.

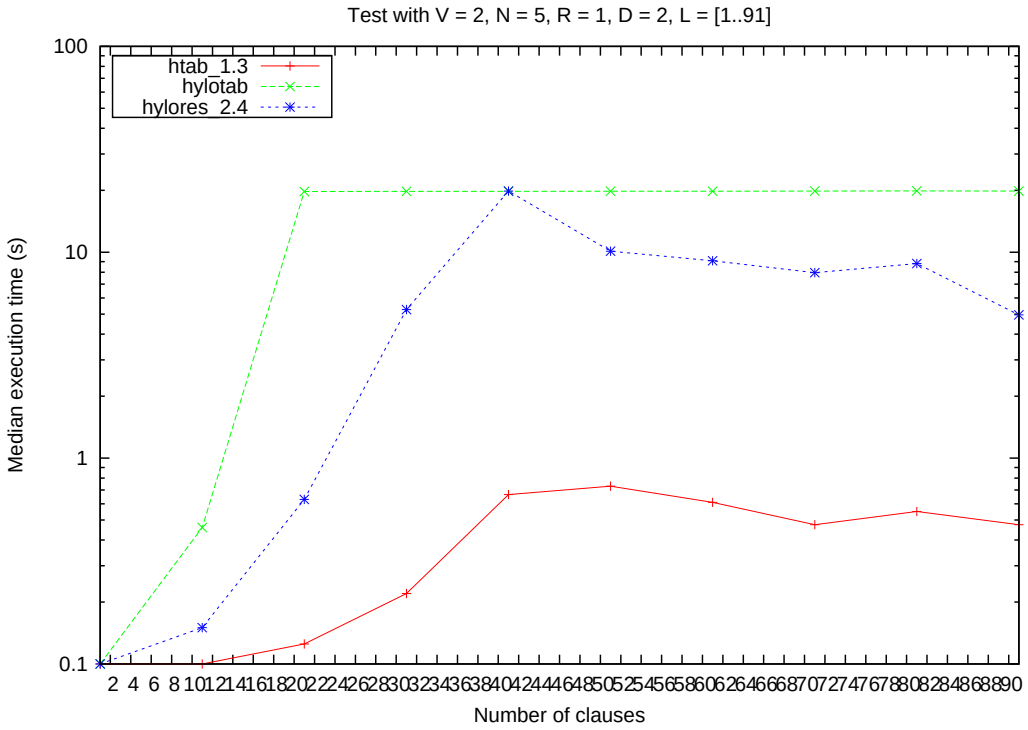

Fig. 3. Median time of execution between HyLoTab, HTab and HyLoRes ($\mathcal{H}(@)$)

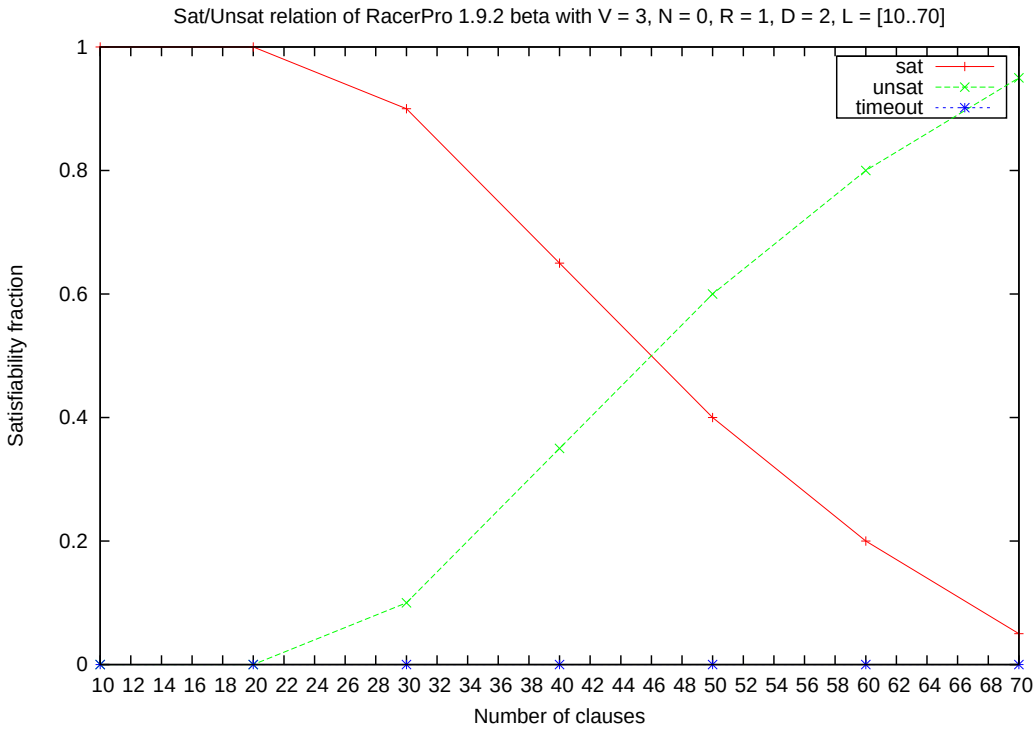

Fig. 4. SAT/UNSAT/Timeout repartition of the formulas for RacerPro ($\mathcal{M}(\mathsf{A})$)

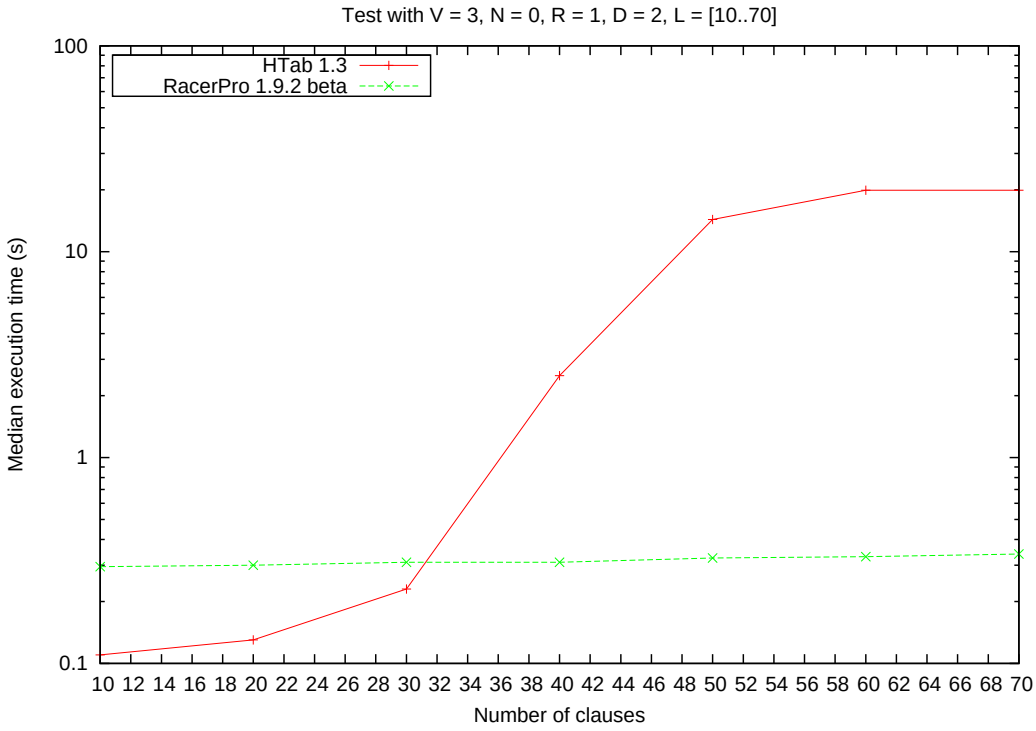

Fig. 5. Median time of execution between HTab and RacerPro ($\mathcal{M}$(A))

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

Executing HTab on these formulas is done with this call:

```
$ htab -f test.frm

The formula is satisfiable.
(final statistics)
begin
-----------------------------------
Closed branches: 9056
-----------------------------------
end
Elapsed time: 2.308143
```

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

7 Conclusion

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

The implementation for the handling of basic hybrid formulas of $\mathcal{H}(@)$ is efficient, but we are still at an early stage of implementation for the blocking condition for the universal modality. Moreover, we have not yet used some optimisations of the basic tableaux algorithm which are standards in state-of-the-art tableaux based provers (e.g., model caching).

Once this hybrid logic is tamed, our next goal is to handle 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. Proceedings of 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] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[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] Edith Hemaspaandra. The price of universality. *Notre Dame Journal of Formal Logic*, 37(2):174–203, 1996.

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

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

[11] Edith Spaan. *Complexity of Modal Logics*. PhD thesis, Department of Mathematics and Computer Science, University of Amsterdam, 1993.

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