KED: A Deontic Theorem Prover

Alberto Artosi*, Paola Cattabriga**, Guido Governatori**

*Dipartimento di Filosofia, Università di Bologna, via Zamboni,38, 40126 Bologna (Italy), Fax:051-258326. **CIRFID, Università di Bologna, via Galliera, 3, 40126 Bologna (Italy), Fax:051-260782, E-mail: governat@cirfid.unibo.it, paola@cirfid.unibo.it.

1 Introduction

Deontic logic (DL) is increasingly recognized as an indispensable tool in such application areas as formal representation of legal knowledge and reasoning, formal specification of computer systems and formal analysis of database integrity constraints. Despite this acknowledgement, there have been few attempts to provide computationally tractable inference mechanisms for DL (most notably [Bel87], [McC83], [McC86], [Sus87]). In this paper we shall be concerned with providing a computationally oriented proof method for standard DL (SDL), i.e., normal systems of modal logic with the usual possible-worlds semantics ([Aq87], [Ch80], [Han65]). Because of the natural and easily implementable style of proof construction it uses, this method seems particularly well-suited for applications in the AI and Law field, and though in the present version it works for SDL only, it forms an appropriate basis for developing efficient proof methods for more expressive and sophisticated extensions of SDL. The content of the paper is as follows. In Section 2, we briefly introduce SDL together with the logical notation being used. In Section 3, we describe the theorem proving system *KED*. In Sections 4 and 5, we present *KED* method of proof search. In the last section, we provide a sample of the *KED* Prolog implementation and give an example output of the program.

2 Preliminaries

A system of SDL is a logic (set of axioms scheme and inference rules) based on a standard modal language consisting of a denumerable set of propositional variables and the primitive logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, P, O for negation, conjunction, disjunction, conditionality, permission and obligation, respectively. We shall use the letters A, B, C,... to denote arbitrary formulas of this language. A system of SDL will be denoted by *L*. We define an *L-model* to be a triple <W, R, *v*> where W is a non-empty set (the set of "possible worlds"), R is a binary relation on W (the "accessibility relation" between the "actual" world and its deontically ideal versions), and *v* is a mapping from $W \times S$ to $\{T,F\}$ where S is the set of all formulas of our present language. As usual, the notion of *L*-model appropriate for the logic *L* can be obtained by restricting R to satisfy the conditions associated with *L*. The following table gives a complete picture of the systems of SLD we shall consider.

L	Definition	Condition on R
OK	$PC \cup \{O(A \rightarrow B) \rightarrow (OA \rightarrow OB)\}$	no condition
D	$OK \cup \{OA \rightarrow PA\}$	idealisation
D4	$D \cup \{OA \rightarrow OOA\}$	idealisation, transitive
DB	$D \cup \{A \rightarrow OPA\}$	idealisation, symmetric
D5	$D \cup \{PA \rightarrow OPA\}$	idealisation, euclidean
D45	$D4 \cup \{PA \rightarrow OPA\}$	idealisation, transitive, euclidean

Table 1: Systems of SDL and their associated conditions.

To complete the definition, all these logics include *modus ponens* and the rule of O-necessitation (if we have already proved A, then we can infer OA). The "idealisation" condition corresponds to the obvious requirement that every world in W has at least an ideal version.

By a *signed formula* (*S-formula*) we shall mean an expression of the form *S*A where A is a formula and $S \in \{T,F\}$. Thus *T*A if $v(x,A) = V$ and *F*A if $v(x,A) = F$ for some *L*-model $\langle W,R,v \rangle$ and $x \in W$. We shall denote by X, Y, Z arbitrary signed formulas. By the *conjugate* X^C of a signed formula X we shall mean the result of changing *S* to its opposite (thus *T*A is the conjugate of *F*A and *F*A is the conjugate of *T*A). Two *S*-formulas X, Z such that $Z = X^C$ will be called *complementary*. For ease of exposition we shall use Smullyan-Fitting's "α,β,ν,π" unifying notation that classifies *S*-formulas as shown in the following table.

α	α_1	α_2	β	β_1	β_2
*T*A∧B	*T*A	*T*B	*F*A∧B	*F*A	*F*B
*F*A∨B	*F*A	*F*B	*T*A∨B	*T*A	*T*B
*F*A→B	*T*A	*F*B	*T*A→B	*F*A	*T*B
T¬A	*F*A	*F*A	*F*¬A	*T*A	*T*A

ν	ν_0	π	π_0
TOA	*T*A	*T*PA	*T*A
FPA	*F*A	*F*OA	*F*A

Table 2: Classification of signed formulas according to Smullyan-Fitting's unifying notation.

3 The system *KED*

In this section we describe the proof system *KED*. The key features of *KED* are outlined as follows.

3.1 Label formalism

Let $\Phi_C = \{w_1, w_2, w_3,...\}$ and $\Phi_V = \{W_1, W_2, W_3,...\}$ be two (non empty) sets of "world" symbols, respectively constant and variables. We define the set $\Im$ of "world" *labels* in the following way:

$\Im = \bigcup_{1\leq n} \Im_i$ where $\Im_i$ is:

$\Im_1 = \Phi_C \cup \Phi_V$

$\Im_2 = \Im_1 \times \Phi_C$

$\Im_{n+1} = \Im_1 \times \Im_n$.

That is a label i is either a constant "world" symbol, or a "world" variable, or a "path" term (k',k) where (a) $k' \in \Phi_C \cup \Phi_V$ and (b) $k \in \Phi_C$ or $k = (m',m)$ where (m',m) is a label. Intuitively, we may think of a label $i \in \Phi_C$ as denoting a world, and a label $i \in \Phi_V$ as denoting a set of worlds in some *L*-model. A label $i = (k',k)$ may be viewed as representing a path from k to a (set of) world(s) k' accessible from k. For example, the label (W_1,w_1) represents a path which takes us to a set W_1 (the set of worlds accessible from the initial world w_1); $(w_2,(W_1,w_1)))$ represents a path which takes us to a world w_2 accessible by any world accessible from w_1, (i.e., accessible by the subpath (W_1,w_1)) and so on (notice that the labels are read from right to left). For any label $i = (k',k)$ we shall call k' the *head* of i, k the *body* of i, and denote them by $h(i)$ and $b(i)$ respectively. Notice that these notions are recursive: if $b(i)$ denotes the body of i, then $b(b(i))$ will denote the body of $b(i)$, $b(b(b(i)))$ will denote the body of $b(b(i))$, and so on. For example, if i is $(w_4,(W_3,(w_3,(W_2,w_1))))$, then $b(i) = (W_3,(w_3,(W_2,w_1)))$, $b(b(i)) = (w_3,(W_2,w_1))$, $b(b(b(i))) = (W_2,w_1)$, $b(b(b(b(i)))) = w_1$. We call each of $b(i)$, $b(b(i))$, etc., a *segment* of i. Let $s(i)$ denote any segment of i (obviously, by definition every segment $s(i)$ of a label i is a label); then $h(s(i))$ will denote the head of $s(i)$. For any label i, we shall denote the *length* of i by $l(i)$, where $l(i) = n \Leftrightarrow i \in \Im_n$.We shall call a label i *restricted* if $h(i) \in \Phi_C$, otherwise we shall call it *unrestricted*.

3.2 Basic unifications

We define a substitution in the usual way as a function $\sigma: \Phi_V \rightarrow \Im^-$ where $\Im^- = \Im - \Phi_V$. Following convention we denote by $i\sigma$, $k\sigma$ the result of applying σ to labels i and k. If $i\sigma = k\sigma$ we shall say that σ *unifies* i and k. Two labels i, k will be said σ-*unifiable* if there is a substitution σ that unifies i and k. In the following we shall use $(i,k)\sigma$ to denote both that i and k are σ-unifiable and the result of their unification. On this basis we define several specialised, logic-dependent notions of σ-unification. In particular, we define the notion of two labels i, k being σ^{OK}-, σ^{D}-, σ^{D4}- and σ^{D5}-unifiable in the following way:

$(i,k)\sigma^{OK} = (i,k)\sigma \Leftrightarrow$
(i) at least one of i and k is restricted, and
(ii) for every $s(i)$, $s(k)$ such that $l(s(i)) = l(s(k))$, $(s(i),s(k))\sigma^{OK}$

$(i,k)\sigma^{D} = (i,k)\sigma$

$(i,k)\sigma^{D4} = h(k) \times (h(b(k)) \times (...\times (t^*(k) \times (i,s(k))\sigma^{D})...)) \Leftrightarrow$
$l(i) \leq l(k)$, $h(i) \in \Phi_V$ and $(i,s(k))\sigma^{D}$,

or

$(i,k)\sigma^{D4} = h(i) \times (h(b(i)) \times (...\times (t^*(i) \times (s(i),k)\sigma^{D})...)) \Leftrightarrow$
$l(k) \leq l(i)$, $h(k) \in \Phi_V$ and $(s(i),k)\sigma^{D}$

where $t^*(k)$ (resp. $t^*(i)$) denotes the element of k (resp. i) which immediately follows $s(k)$ (resp. $s(i)$).

$$(i,k)\sigma^{D5} = (h(i),h(k))\sigma \times (b(b(i)),b(k))\sigma^{L})) \Leftrightarrow$$
$$(h(i),h(k))\sigma \text{ and } (b(b(i)),b(k))\sigma^{L} \text{ for } l(k) \leq l(i) \text{ or } h(b(k)) \in \Phi_C,$$

or

$$(i,k)\sigma^{D5} = (h(i),h(k))\sigma \times (b(i),b(b(k))\sigma^{L})) \Leftrightarrow$$
$$(h(i),h(k))\sigma \text{ and } (b(i),b(b(k))\sigma^{L} \text{ for } l(i) \leq l(k) \text{or } h(b(i)) \in \Phi_C,$$

where

$$\sigma^{L} = \begin{cases} \sigma^{D} \text{ or } \sigma^{D5} & \text{if } l(i) = l(k) \\ \sigma^{D5} & \text{if } l(i) \neq l(k) \end{cases}$$

The above notions are meant to mirror the conditions on R in the various L-models. Thus the notions of σ^{OK}-and σ^{D}-unification are related to the idealisation condition. For example, $(w_2,(W_1,w_1))$, $(W_3,(W_2,w_1))$ are σ^{D}-unifiable but not σ^{OK}-unifiable (since the segments (W_1,w_1), (W_2,w_1) are not σ^{OK}-unifiable by condition (i) of the above definition). The reason is that in the "non idealisable" logic *OK* the "denotations" of W_1 and W_2 may be empty (i.e., there can be no worlds accessible from w_1), while in the "idealisable" logic *D* they are not empty, which makes them to be unifiable "on" any constant. For the notion of σ^{D4}-unification take for example $i = (W_3,(w_2,w_1))$ and $k = (w_5,(w_4,(w_3,(W_2,w_1))))$. Here $s(k) = (w_3,(W_2,w_1))$. Then i and k σ^{D4}-unify to $(w_5,(w_4,(w_3,(w_2,w_1))))$ since $((W_3,(w_2,w_1)),(w_3,(W_2,w_1)))\sigma^{D}$. This intuitively means that all the worlds accessible from a subpath $s(k)$ of k are accessible from any path i which turns out to be identical with $s(k)$. Similar intuitive motivations hold for the notion of σ^{D5}-unification.

3.3 Reductions

For $X = 4, B$ we define the *X-reduction*, $r_X(i)$, of a label i to be a function r_X: $\Im \rightarrow \Im$ determined as follows:

$$r_4(i) = \begin{cases} h(i), b(b(i)) \\ i, i \text{ unrestricted and } l(i) \leq 3 \end{cases} \qquad r_B(i) = \begin{cases} b(b(i)), i \text{ unrestricted and } l(i) \geq 3 \\ (h(i), r_B(b(i))), i \text{ restricted} \end{cases}$$

The notion of X-reduction holds for the logics whose associated conditions are transitivity and symmetry. As an intuitive explanation, we may think of the X-reduction of a label i as the deletion of "irrelevant" steps from the path represented by i. Thus for example the *4*-reduction (w_2,w_1) of the label $(w_2,(W_1,w_1))$ amounts to deleting the step to an arbitrary world (in the set) W_1 in the path from w_1 to a world w_2 accessible from all worlds accessible from w_1 since if R is constrained to satifsy transitivity, then this step turns out to be irrelevant (w_2 is accessible from w_1 for all W_1 accessible from w_1).

3.4 General unification

We are now able to define what it means for two labels i, k to be σ_L-*unifiable* for L = *OK, D, D4, DB, D5, D45*:

(i) $(i,k)\sigma_L = (i,k)\sigma^*$, where

for $l(i) = l(k)$

$\sigma^* = \sigma^{OK}$, if $L = OK$
$\sigma^* = \sigma^{D}$, if $L = D, D4, DB, D5, D45$
$\sigma^* = \sigma^{D5}$, if $L = D5, D45;$

for $l(i) \neq l(k)$

$\sigma^* = \sigma^{D4}$, if $L = D4$
$\sigma^* = \sigma^{D5}$, if $L = D5, D45$.

(ii) $(i,k)\sigma_L = (i,r_X(k))\sigma^*$, or
$(r_X(i),k)\sigma^*$, or
$(r_X(i),r_X(k))\sigma^*$,

where

$X = 4$, if $L = D4, D45$
$X = B$, if $L = DB$.

and

$\sigma^* = \sigma^{D}$, if $L = D4, DB, D5, D45$
$\sigma^* = \sigma^{D5}$, if $L = D5, D45$.

Notice that in this way all the obvious inclusions among the logics considered are preserved.

3.5 Rules of inference

The rules of *KED* will be defined for pairs X,*i* where X is a signed formula and *i* is label. We shall call any pair X,*i* a *labelled signed formula* (*LS-formula*). Two *LS*-formulas X,*i*, Z,*k* such that $Z = X^C$ and $(i,k)\sigma_L$ are called σ_L-*complementary*. The following inference rules hold for all the logics we are considering (*i*, *i'*, *k* stand for arbitrary labels).

$$\frac{\alpha,i}{\alpha_1,i} \qquad \frac{\alpha,i}{\alpha_2,i} \qquad \frac{\beta,i \quad \beta^C{}_1,k}{\beta_2,(i,k)\sigma_L} \qquad \frac{\beta,i \quad \beta^C{}_2,k}{\beta_1,(i,k)\sigma_L} \qquad [(i,k)\sigma_L]$$

$$\frac{\nu,i}{\nu_0,(i',i)} \quad [(i',i) \text{ unrestricted and } i' \text{ new}] \qquad \frac{\pi,i}{\pi_0,(i',i)} \quad [(i',i) \text{ restricted and } i' \text{ new}]$$

$$\frac{}{\mathrm{X},i \quad \mathrm{X}^{\mathrm{C}},i}\,PB \quad [i \text{ restricted}] \qquad\qquad \frac{\mathrm{X},i \quad \mathrm{X}^{\mathrm{C}},k}{\times\,(i,k)\sigma_L}\,PNC \quad [(i,k)\sigma_L]$$

Here the α-rules are just the usual linear branch-expansion rules of the tableau method, while the β-rules correspond to such common natural inference patterns as *modus ponens*, *modus tollens*, etc. The rules for the modal operators bear a not unexpected resemblance to the familiar quantifier rules of the tableau method. "*i'* new" in the proviso for the ν- and π-rule obviously means: *i'* must not have occurred in any label yet used. Notice that in all inferences via an α-rule the label of the premise carries over unchanged to the conclusion, and in all inferences via a β-rule the labels of the premises must be σ_L-unifiable, so that the conclusion inherits their unification. (The underlying intuitive motivation is that *LS*-formulas whose labels are σ_L-unifiable turns out to be true (false) at the same world(s) relative to the associated conditions on R). *PB* (the "Principle of Bivalence") and *PNC* (the "Principle of Non-Contradiction") are "structural" rules. *PB* represents the (*LS*-version of the) semantic counterpart of the cut rule of the sequent calculus (intuitive meaning: a formula A is either true or false in any *given* world, whence the requirement that *i* be restricted). *PNC* corresponds to the familiar branch-closure rule of the tableau method, saying that from a contradiction of the form (occurrence of a pair of σ_L-complementary *LS*-formulas) X,*i*, X^{C},*k* on a branch we may infer the closure ("$\times$") of the branch. The (*i*,*k*) σ_L in the "conclusion" of *PNC* means that the contradiction holds "in the same world".

It can be proved ([AG93],[AG94]) that the above rules give a sound and complete system for a wide variety of normal modal logics.

4 Proof search

As usual with refutation methods, a proof of a formula A of *L* consists of attempting to construct a countermodel for A by assuming that A is false in some arbitrary *L*-model. Every successful proof discovers a contradiction in the putative countermodel. In this section we describe an algorithm which does this job and that can be easily implemented in Prolog (see Section 7 below). The following definitions are extensions to the modal case of those given for the classical case in [DM94].

By a *KED-tree* we mean a tree generated by the inference rules of *KED*. A branch τ of a *KED*-tree will be said to be σ_L-*closed* if it ends with an application of *PNC*. A *KED*-tree will be said to be σ_L-*closed* if all its branches are σ_L-closed. A *L*-proof of a formula A is a σ_L-closed *KED*-tree starting with *F*A,*i*. Given a branch τ of a *KED*-tree, we shall call a *LS*-formula X,*i* *E-analysed in* τ if either (i) X is of type α and both α_1,*i* and α_2,*i* occur in τ; or (ii) X is of type β and one of the following conditions is satisfied: (a) if $\beta^{\mathrm{C}}{}_1$,*k* occurs in τ and (*i*,*k*)σ_L, then also β_2,(*i*,*k*)σ_L occurs in τ, (b) if $\beta^{\mathrm{C}}{}_2$,*k* occurs in τ and (*i*,*k*)σ_L, then also β_1,(*i*,*k*)σ_L occurs in τ; or (iii) X is of type ν and ν_0,(*i'*,*i*) occurs in τ for some i' $\in \Phi_{\mathrm{V}}$ not previously occurring in τ, or (iv) X is of type π and π_0,(*i'*,*i*) occurs in τ for some i' $\in \Phi_{\mathrm{C}}$ not previously occurring in τ. We shall call a branch τ of a *KED*-tree *E-completed* if every *LS*-formula in it is *E*-analysed and there are no complementary formulas which are not σ_L-complementary. Finally, we shall call a *LS*-formula X,*i* of type β *fulfilled in a branch* τ if

either β_1,i' or β_2,i' occur in τ, where either (i) $i' = i$, or (ii) i' is obtained from i by instantiating $h(i)$ to a constant not occurring in i, or (iii) $i' = (i,k)\sigma_L$ for some $\beta^C{}_i,k$ (i = 1,2) such that $(i,k)\sigma_L$. We shall say that a branch τ of a *KED*-tree is *completed* if it is both *E*-completed and all the *LS*-formulas of type β in it are fulfilled. We shall call a *KED*-tree *completed* if every branch is completed. Let us denote by Λ (Lambda) the set of *LS*-formulas which occur non analysed, by Δ (Delta) any branch, and by $\mathscr{L}$ the set of labels. The *KED* algorithm runs as follows (the quotations in brackets refer to the Prolog implementation in Section 7). To prove a formula A of *L*

STEP 0. Assign to A an arbitrary constant label *i*, and put SA,i in Δ and i in $\mathscr{L}$.

STEP 1 (cke 1). If a pair of σ_L-complementary *LS*-formulas occurs in Δ, then the tree is σ_L-closed and A is a theorem of *L*.

STEP 2 (cke 3). Delete all literals from Δ. If Δ is empty, then the tree is completed.

STEPS 3, 4 (cke 5,6). For each formula π,i (ν,i) in Δ (i) generate a new restricted (unrestricted) label (i',i) and add it to $\mathscr{L}$; (ii) add $\pi_0,(i',i)$ $(\nu_0,(i',i))$ to Λ; and (iii) delete π,i (ν,i) from Δ.

STEP 5 (cke 7). For each formula α,i in Δ, (i) add α_1,i, α_2,i to Δ; (ii) delete α,i from Δ; and (iii) add α,i to Λ.

STEP 6 (cke 8). For each formula β,i in Δ, such that either β_1,k or β_2,k is in $\Delta \cup \Lambda$ and $(i,k)\sigma_L$, (i) delete β,i from Δ, and (ii) add β,i to Λ.

STEP 7 (cke 9,10). For each formula β,i in Δ such that either $\beta^C{}_1,k$ or $\beta^C{}_2,k$ is in $\Delta \cup \Lambda$ and $(i,k)\sigma_L$ for some label k, (i) add $\beta_2(i,k)\sigma_L$ or $\beta_1(i,k)\sigma_L$ to Δ; (ii) delete β,i from Δ; (iii) add the labels resulting from the σ_L-unification to $\mathscr{L}$; and (iv) add β,i to Λ.

STEP 8.1 (cke 11). For each formula β,i in Δ, if $\Delta \cup \Lambda$ does not contain formulas $\beta^C{}_1,k$ such that i, k are not σ_L-unifiable, then form sets $\Delta_1 = \Delta \cup \beta_1,m$ and $\Delta_2 = \Delta \cup \beta^C{}_1,m$ (where $(i,m)\sigma_L$, and m is a given restricted label).

STEP 8.2 (cked 12). For each formula β,i in Δ, if $\Delta \cup \Lambda$ does not contain formulas $\beta^C{}_2,k$ such that i, k are not σ_L-unifiable, then form sets $\Delta_1 = \Delta \cup \beta_2,m$, and $\Delta_2 = \Delta \cup \beta^C{}_2,m$ (where $(i,m)\sigma_L$, and m is a given restricted label).

Remark 1: The steps 8.1 and 8.2 are logic and label dependent. This mean that if the label of X is restricted, its immediate signed subformulas have the same label as X, otherwise we have to deal with two cases: a) search whether $\mathscr{L}$ contains restricted labels which σ_L-unify with the label of X; if so the rule is applied to all such labels; b) if *L* is an idealisable logic then, if the search fails, $h(i)$ is instantiated to a new constant label not previously occurring.

STEP 9 (cke14). If Λ contains two complementary but not σ_L-complementary formulas, search in $\mathscr{L}$ for restricted labels which σ_L-unify with both the labels of the complementary formulas; if we find such labels then the tree is closed and A is a theorem of *L*.

STEP 10 (cke15). If Λ contains two complementary but not σ_L-complementary formulas , search in $\mathscr{L}$ for restricted labels which σ_L-unify with both the labels of the complementary formulas; if we do not find such labels then the tree is completed and A is not a theorem of *L*.

This procedure is based on the procedure for canonical *KED*-trees. A *KED*-tree is said to be canonical iff the applications of 1-premise rule come before the applications of 2-premise rules, which preceed the applications of the 0-premise rule. The following theorems state some interessing properties of canonical *KED*-trees:

THEOREM 1. A canonical *KED*-tree always terminates.

THEOREM 2. A *KED*-tree for a formula A of *L* is closed iff the canonical *KED*-tree for A is closed.

Theorem 1 follows from the fact that at each step there are at most a finite number of new *LS*-formulas of less complexity, and that the number of labels which can occur in the *KED*-tree for a formula A (of *L*) is limited by the number of modal operators in A. Theorem 2 follows from the fact that a canonical *KED*-tree is a *KED*-tree and that a *KED*-tree explores all the possibile alternatives that can imply closure (for detail see [ACG94a]).

Remark 2: It should be noticed that in the above procedure *PB* is applied *only* to immediate signed subformulas of *LS*-formulas of type β which occur (unfulfilled) in the chosen branch, and *only* when the branch has been *E*-completed, *i.e.*, when the *E*-rules are no further applicable. Such a *restricted* use of the cut rule removes from the search space the redundancy generated by the standard tableau branching rules. Indeed it is easy to see that the given procedure makes all choices in such a way that at each step of proof search the search space is as small as possible, while preserving the subformula property of proofs (see [DM94]).

5 An example

We illustrate the *KED*-based search procedure with the help of an example. The following is a *D*-proof of the formula (PA∨OB)→P(A∨B).

(1) *F*(PA∨OB)→P(A∨B), w_1
(2) *T*PA∨OB, w_1
(3) *F*P(A∨B), w_1
(4) *F*A∨B, (W_1,w_1)
(5) *F*A, (W_1,w_1)
(6) *F*B, (W_1,w_1)

(7) *T*PA, w_1 | (8) *F*PA, w_1
(9) *T*A, (w_2,w_1) | (10) *T*OB, w_1
× (w_2,w_1) | (11) *T*B, (W_2,w_1)
× (w_2,w_1)

The steps leading to the nodes (1)-(6) are straightforward. At this stage, to complete the branch we pick out the only *LS*-formula of type β which is not yet fulfilled in it, i.e., (2), and apply *PB* so that the resulting *LS*-formulas are (7) and (8). At this point an application of π-rule σ_D-closes the left branch ((5) and (9) are obviously σ_D-complementary by condition (i) of the above definition). To make the right branch *E*-completed, we choose the only *LS*-formula of type β which is not yet analysed in it, i.e., (2), and try to σ_D-unify its label with the label of $\beta^C{}_1$,*k* (*i.e.*, (8)). Since this unification succeeds we are allowed to derive β_2,(*i*,*k*)σ_D at the node (10). Now an application of ν-rule σ_D-closes the branch. The resulting *KED*-tree is thus σ_D-closed. Notice that (6) and (11), are σ_D- but not σ_{OK}-complementary (their labels are obviously not σ^{OK}-unifiable, and thus not σ_{OK}-unifiable). Then this *KED*-tree constitutes a *D*- (and, of course, a *D4*-, *DB*-) proof, but not a *OK*-proof of the given formula.

6 Final remarks

Let us conclude with some comments on *KED* and related systems. In our opinion *KED* has several advantages over most automated theorem proving systems for non-classical logics currently available. Here we mention only a few. In contrast with both clausal and non clausal resolution methods ([AEH90], [AM86], [Cia86], [EF89], [Far85], [Far86], [Cha87]), and in general "translation-based" methods ([AE92], [Ohl87], [Ohl89], [Ohl91]), *KED* requires no preprocessing of the input formulas and provides a simple and uniform treatment of a wide class of normal modal logics ([AG94], [ACG94b]). From this perspective it is similar to sequent or tableau proof methods ([Fit88], [GD88], [JR89], [Wol85]), which avoid *ad hoc* manipulation of the modal formulas and can be easily extended to a wide variety of non-classical logics. Nevertheless, it is well-known that sequent/tableau inference techniques are affected by considerable redundancies which prevent the development of computationally efficient proof search methods. *KED* is based on Mondadori's ([Mon88]) classical proof system *KE* which, though being tableau-like, has been proved ([DM94]) to offer many computational advantages over standard tableau method, including considerable gain in efficiency and conciseness. The critical feature of *KED* is that it developed as a *labelled* system, similar in spirit to D'Agostino and Gabbay [DG93]) tableau extensionwith labels. The idea of using a label scheme to bookkeep "world" paths in modal theorem proving is not new, going back at least to [Fi66]. Similar, or related, ideas are found in [Fit72], [Fit83], [Tap87] and [Wri85] and, more recently, in [Cat91], [JR89], [Wal90] and also in the "translation" tradition of [Ae92], [Ohl89], [Ohl91], [Ohl93]. As in Wallen's ([Wal90], [Gen93]) matrix proof method, *KED*'s label scheme allows the modal operators to be dealt with using a specialized, logic-dependent unification algorithm to overcome the non-permutability of the usual tableau (and resolution) modal rules. However, unlike the Wallen's method (a generalization of Bibel's classical connection method) *KED* implements *directly* familiar, natural inference patterns, and so it appears to provide an adequate basis for combining both efficiency and naturalness. In effect, we believe that *KED* lends itself well to both interactive and AI applications. As its Prolog implementation (see Section 7 below) has shown, *KED* method of proof is simple and easy to implement - but simple enough to be used without a machine. These are not, however, the only advantages of *KED*'s label unification scheme. For example, the index formalism of Jackson and Reichgelt's ([JR89]) sequent-resolution based proof system is almost identical, but the unification algorithm used to resolve

complementary formulas in the various modal logics does not work for the non-idealisable *K* logics. Further advantages of *KED* label formalism are that (i) it avoids loop-checking and reduplication (see Section 4 above); (ii) it supports a deduction method closely related to the semantics of modal operators; and (iii) it works for modal logics whose characteristic semantic properties have no first-order characterization, such as the Gödel-Löb logic of provability [Boo79].

In this paper we have been concerned with SDL. This may be seen as a major limitation, since it is currently held that SDL fails to provide a framework suited for applications in the AI and Law field, its use being limited to very general and unproblematic features of normative language and reasoning. Nervertheless, the method for automated deduction in SDL we presented in this paper is sufficiently generic and flexible to provide an appropriate algorithmic proof framework for deontic logics of greater richness and complexity. For example, it can be easily extended to multi modal logics (e.g., to the deontic logic of the Jones-Pörn type [JP85]), by simply introducing several sorts of "worlds", both constants and variables, or by building indexed labels where an index tells us what kind of world is denoted by a label, and the way you get there. Moreover several specialised substitutions have to be defined in order to conform to the constraints of the logic to be dealt with (of course, these tecniques can be combined).

7 A sample of a *KED* Prolog program

The following Prolog implementation of *KED* is based on that of the classical proof system *KE* by Pitt and D'Agostino (see [DP94]).
KED selects a deontic logic *L* and it runs the *KED* rules for that logic. If the input formula is a theorem of *L* the program will output the entire reserch path with the final answer "closed" "theorem of *L*", and *KED* stops to run. If the input formula is not *L*-satisfiable, *KED* selects another logic and try to find the solution. The complete Prolog version of *KEM* ("*M*" for "modal") can prove formulas of the following logics *K*, *D*, *T*, *K4*, *D4*, *S4*, *K5*, *D5*, *KB*, *DB*, *B*, *K4B*, *K45*, *D45*, and *S5* (see [ACG94a]). The ":" operator attaches labels to formulas and "labeltree()" records the labels. In this way the labels have a semantical control concurrent function over the syntactical inference rules. In what follows, ->, +, &, ˜, $, @ denotes $\rightarrow$, $\vee$, $\wedge$, $\neg$, P, O respectively.

```
Quintus implementations on SparcStation 10.
/*      KED   - Deontic Theorem Proving   -  by Alberto Artosi, Paola Cattabriga,and Guido Governatori.*/

:- ensure_loaded(library(basics)).
:- ensure_loaded(library(occurs)).
:- ensure_loaded(library(term_depth)).

:- op(800,xfx,:).
:- op(600,xfy,->).
:- op(550,xfy,+).
:- op(450,xfy,&).
:- op(300,fy, ~).
:- op(240,fy,$).
:- op(240,fy,@).

all_log(SetOfFormula):-
findall(SetOfFormula,pr(SetOfFormula),_).

pr(SetOfFormula):- logic(L),
                   pr1(SetOfFormula),
                   retract(log(L)).

pr(L,SetOfFormula):- logic(L),
                   pr1(SetOfFormula),
                   retractall(log(LL)).

pr1(SetOfFormula):-   statistics(runtime,[T1|_]),
                   log(L),
                   assert(labeltree([i(w(1),w(1))])),
                   label(SetOfFormula,SLF),
                   cke(SLF,[],Result), !,
                   statistics(runtime,[T2|_]),
                   T is T2 - T1,
                   write(Result), write('  in  '), write(L), nl,
                   write('   in   '), write(T), write('  msecs.'), nl,nl,
```

```
        retract(labeltree(_)).

cke(_,Lambda,unsatisfiable):-
        member(I: ~A,Lambda),
        member(K: A,Lambda),
        log(L),
        unifylow(L,I,K,_), write(I: ~A), write(', '),
        write(K: A),
        write(' unify in  '), write(L), nl,
        write(Lambda),
        write('***closed'), nl, !.

cke([],Lambda,satisfiable):-
        write(Lambda),
        write('***completed'), nl, !.

cke(Delta,Lambda,Result):-
        append(H,[F|T],Delta),
        literal(F),
        append(H,T,Delta1),
        write(Delta), write( <---> ), write(Lambda), nl,
        write('literal'), nl,
        cke(Delta1,[F|Lambda],Result).

cke(Delta,Lambda,Result):-
        append(H,[I: ~(~A)|T],Delta),
        append(H,[I:A|T],Delta1),
        write('double negation elimination'), nl,
        cke(Delta1,[I: ~(~A)|Lambda],Result).

cke(Delta,Lambda,Result):-
        append(H,[I: F|T],Delta),
        type_ni(I: F,K: A),  genv(I,K),
        labeltree(K,V),
        append(H,[K: A|T],Delta1),
        write(Delta), write( <---> ), write(Lambda), nl,
        write('ni elimination'), nl,
        cke(Delta1,[I: F|Lambda],Result).

cke(Delta,Lambda,Result):-
        append(H,[I: F|T],Delta),
        type_pi(I: F,K : A), genc(I,K),
        labeltree(K,V),
        append(H,[K: A|T],Delta1),
        write(Delta), write( <---> ), write(Lambda), nl,
        write('pi elimination'), nl,
        cke(Delta1,[I: F|Lambda],Result).

cke(Delta,Lambda,Result):-
        append(H,[F|T],Delta),
        type_alpha(F,A1,A2),
        append(H,[A1,A2|T],Delta1),
        write(Delta), write( <---> ), write(Lambda), nl,
        write('alpha elimination'), nl,
        cke(Delta1,[F|Lambda],Result).

cke(Delta,Lambda,Result):-
        append(H,[I: F|T],Delta),
        type_beta(I: F,K: B1,R:B2),
        append(Delta,Lambda,DuL), log(L),
        ((member(K: B1,DuL), unifylow(L,I,K,_),
        write(I: F),
        write(', '),write(K: B1),
        write(' unify in  '), write(L), nl);
        (member(R: B2,DuL), unifylow(L,I,R,_),
        write(I: F),
        write(', '),write(R: B2),
        write(' unify in  '), write(L), nl)),
        append(H,T,Delta1),
        write(Delta), write( <---> ), write(Lambda), nl,
        write('beta semplification 1'), nl,
        cke(Delta1,[I: F|Lambda],Result).

cke(Delta,Lambda,Result):-
        append(H,[X: F|T],Delta),
        type_beta(X: F,B1,Q: B2),
        append(Delta,Lambda,DuL),
        complement(B1,Y: B1c),
        member(Y: B1c,DuL), log(L),
        unifylow(L,X,Y,R), labeltree(R,V),
        append(H,[R: B2|T],Delta1),
        write(X: F), write(', '),write(Y: B1c),
        write(' unify in  '), write(L), nl,
        write(Delta), write(  <---> ), write(Lambda), nl,
        write('beta elimination 1'), nl,
        cke(Delta1,[F|Lambda],Result).

cke(Delta,Lambda,Result):-
        append(H,[X: F|T],Delta),
        type_beta(X: F,Q: B1,B2),
        append(Delta,Lambda,DuL),
        complement(B2,Y: B2c),
        member(Y: B2c,DuL), log(L),
        unifylow(L,X,Y,R),labeltree(R,V),
        append(H,[R:B1|T],Delta1),
        write(X: F), write(', '),write(Y: B2c),
        write(' unify in  '), write(L), nl,
        write(Delta), write( <---> ), write(Lambda), nl,
        write('beta elimination 2'), nl,
        cke(Delta1,[F|Lambda],Result).

cke(Delta,Lambda,Result):-
        append(H,[I: F|T],Delta),
        type_beta(I: F,K: B1,_),
        labeltree(V), log(L),
        ((member(i(w(Y),X),V),
        unifylow(L,I,i(w(Y),X),_),i(w(Y),X)= K);
        ((log(d); log(d4); log(d5); log(d45); log(db)),
        memb(i(vw(Z),X1),V), genc(X1,K),
        unifylow(L,I,K,_))),
        write(I: F), write(', '),write(K: B1),
        write(' unify in  '), write(L), nl,
        complement(K: B1,K: B1c),
        append(H,[K: B1,I: F|T],Delta1),
```

```
        append(H,[K: B1c,I: F|T],Delta2),
        write('pb1'), nl,
        write('branch 1'), nl,
        cke(Delta1,Lambda,R1),
        write('branch 2'), nl,
        cke(Delta2,Lambda,R2),
        eval(R1,R2,Result).

cke(Delta,Lambda,Result):-
        append(H,[I: F|T],Delta),
        type_beta(I: F,_,K: B2),
        labeltree(V), log(L),
        ((member(i(w(Y),X),V),
        unifylow(L,I,i(w(Y),X),_),i(w(Y),X)= K);
        ((log(d); log(d4); log(d5); log(d45); log(db)),
        memb(i(vw(Z),X1),V), genc(X1,K),
        unifylow(L,I,K,_))),
        write(I: F), write(', '),write(K: B2),
        write(' unify in  '), write(L), nl,
        complement(K: B2,K: B2c),
        append(H,[K: B2,I: F|T],Delta1),
        append(H,[K: B2c,I: F|T],Delta2),
        write('pb2'), nl,
        write('branch 1'), nl,
        cke(Delta1,Lambda,R1),
        write('branch 2'), nl,
        cke(Delta2,Lambda,R2),
        eval(R1,R2,Result).

cke(_,Lambda,unsatifiable):-
        member(I: ~A,Lambda),
        member(K: A,Lambda),
        labeltree(H),
        member(i(w(Y),X),H), R = i(w(Y),X), log(L),
        unifylow(L,K,R,R1), unifylow(L,I,R,R1),
        write(I: ~A), write(', '), write(K: A),
        write(' unify in  '), write(L), nl,
        write(Lambda),
        write('***closed pb mod'), nl, !.

cke([],Lambda,satisfiable):-
        member(I: ~A,Lambda),
        member(K: A,Lambda),
        labeltree(H),
        member(i(w(Y),X),H), R = i(w(Y),X), log(L),
        \+ unifylow(L,K,R,R1), \+ unifylow(L,I,R,R1),
        write(Lambda),
        write('***completed pb mod'), nl, !.

eval(satisfiable,_,satisfiable).
eval(_,satisfiable,satisfiable).
eval(_,_,unsatisfiable).

literal(I:A):- atom(A).
literal(I: ~A):- atom(A).

complement(I: ~A,K: A).
complement(I: A,K: ~A).

type_alpha(I: ~(A+B),I: ~A,I: ~B).
type_alpha(I: A&B,I: A,I: B).
type_alpha(I: ~(A->B),I: A,I: ~B).

type_beta(I: A+B,J: A,K: B).
type_beta(I: ~(A&B),J: ~A,K: ~B).
type_beta(I: A->B,J: ~A,K: B).

type_ni(I : $ A,K : A).
type_pi(I : @ A,K : A).
type_ni(I : ~ @ A,K : ~A).
type_pi(I : ~ $ A,K : ~A).

label([],[]).
label([T|C],[Y: T|C1]):- labeltree(X),
member(Y,X),label(C,C1).

labeltree(K,V):- labeltree(V), retract(labeltree(V)),
        assert(labeltree([K|V])).

costants(w(1)).
costants(w(N1)):- costants(w(N)), N1 is N+ 1.
variables(vw(1)).
variables(vw(N1)):- variables(vw(N)), N1 is N+ 1.

genc(_,i(w(2),i(w(1),w(1)))):- labeltree([i(w(1),w(1))]).
genc(I,i(w(N),I)):- labeltree(T), costants(w(N)),
        append([A|_],C,T), \+ (sub_term(w(N),A)), !.

genv(_,i(vw(1),i(w(1),w(1)))):- labeltree([i(w(1),w(1))]).
genv(I,i(vw(N),I)):- labeltree(T), variables(vw(N)),
        append([A|_],C,T),  \+ (sub_term(vw(N),A)), !.

logic(L):-  selectlogic(L), assert(log(L)).
selectlogic(k).
selectlogic(d).
selectlogic(d4).
selectlogic(d5).
selectlogic(d45).
selectlogic(db).

/* unification  theory  over  labels  */

unifylow(L,X,Y,Z):- log(L), low(L,X,Y,Z).
low(k,X,Y,Z):- lowunifyk(X,Y,Z).
low(d,X,Y,Z):- unifyd(X,Y,Z).
low(d4,X,Y,Z):- lowunifyd4(X,Y,Z).
low(d5,X,Y,Z):- lowunifyd5(X,Y,Z).
low(d45,X,Y,Z):- lowunifyd45(X,Y,Z).
low(db,X,Y,Z):- lowunifydb(X,Y,Z).

unifyd(vw(N),vw(N1),vw(N2)):- (N >= N1, N2 = N);
        N1 =N2.
unifyd(w(N),vw(N1),w(N)).
unifyd(vw(N1),w(N),w(N)).
```

```
unifyd(w(N),w(N),w(N)).
unifyd(i(A,B),i(C,D),i(E,G)):-  functor(i(A,B),F,N),
        functor(i(C,D),F,N),
        unifyargs(N,i(A,B),i(C,D),i(E,G)).

unifyargs(N,X,Y,T):- N>0, unifyarg(N,X,Y,AT),
        N1 is N - 1,
        functor(T,i,2), arg(N,T,AT),
        unifyargs(N1,X,Y,T).
unifyargs(0,X,Y,T).
unifyarg(N,X,Y,AT):- arg(N,X,AX), arg(N,Y,AY),
        unifyd(AX,AY,AT).

/* reductions and low unifications*/
/*low unification for K, D are equal to high unification.*/

reduct4(i(w(N),w(N1)),i(w(N),w(N1))):- !.
reduct4(i(vw(N),w(N1)),i(vw(N),w(N1))):- !.
reduct4(i(vw(N),i(vw(N1),w(N2))),i(vw(N),i(vw(N1),w(N2)))):- !.
reduct4(i(vw(N),i(w(N1),w(N2))),i(vw(N),i(w(N1),w(N2)))):- !.
reduct4(T1,T2):- compound(T1), arg(1,T1,H1),
arg(2,T1,B1), arg(2,B1,BB1),
        functor(T2,i,2), arg(1,T2,H1), arg(2,T2,BB1).

reductb(i(w(N),w(N1)),i(w(N),w(N1))):- !.
reductb(i(vw(N),w(N1)),i(vw(N),w(N1))):- !.
reductb(i(vw(N),i(vw(N1),w(N2))),i(vw(N),i(vw(N1),w(N2)))):- !.
reductb(i(vw(N),i(w(N1),w(N2))),i(vw(N),i(w(N1),w(N2)))):- !.
reductb(T1,T2):- compound(T1), H = i(vw(N),S),
        sub_term(H,T1), arg(2,S,B1),
        subs(H,T1,B1,T2).

reduct5(i(w(1),w(1)),i(w(1),w(1))):- !.
reduct5(i(vw(N),w(N1)),i(vw(N),w(N1))):- !.
reduct5(T1,T2):- compound(T1),  H = i(w(N), S),
        sub_term(H,T1), arg(2,S,B1),
        subs(H,T1,B1,T2).

reduct5(T1,T2):- compound(T1),  arg(1,T1,H1),
        arg(2,T1,B1), arg(2,B1,BB1),
        functor(T2,i,2), arg(1,T2,H1), arg(2,T2,BB1).

subs(T,T,T1,T1):- !.
subs(_,T,_,T):- atom(T), !.
subs(S,T,S1,T1):- T =..[F|Arg], subslist(S,Arg, S1,Arg1),
        T1 =.. [F|Arg1].

subslist(_,[],_,[]).
subslist(S,[T|Ts], S1,[T1|T1s]):-  subs(S,T,S1,T1),
        subslist(S,Ts,S1,T1s).
lowunifyk(T1,T2,T3):-  unifyk(T1,T2,T3).

lowunifyd4(T1,T2,T3):- arg(1,T1,w(Y)),  arg(1,T2,w(X)),
        X \== Y, !, fail.
lowunifyd4(T1,T2,T3):-  unifyd(T1,T2,T3);
        unifyd4(T1,T2,T3).
lowunifyd4(T1,T2,T3):-  reduct4(T1,R1), T1 \== R1,
        lowunifyd4(R1,T2,T3).
lowunifyd4(T1,T2,T3):-  reduct4(T2,R2), T2 \== R2,
        lowunifyd4(T1,R2,T3).
lowunifyd4(T1,T2,T3):-  reduct4(T1,R1), reduct4(T2,R2),
        T1 \== R1, T2 \== R2, lowunifyd4(R1,R2,T3).

lowunifydb(T1,T2,T3):- arg(1,T1,w(Y)),  arg(1,T2,w(X)),
        X \== Y, !, fail.
lowunifydb(T1,T2,T3):-  unifyd(T1,T2,T3).
lowunifydb(T1,T2,T3):-  reductb(T1,R1), T1 \== R1,
        lowunifydb(R1,T2,T3).
lowunifydb(T1,T2,T3):-  reductb(T2,R2), T2 \== R2,
        lowunifydb(T1,R2,T3).
lowunifydb(T1,T2,T3):-  reductb(T1,R1), reductb(T2,R2),
        T1 \== R1, T2 \== R2, lowunifydb(R1,R2,T3).

lowunifyd5(T1,T2,T3):- unifyd(T1,T2,T3).
lowunifyd5(T1,T2,T3):- unifyd5(T1,T2,T3).
lowunifyd5(T1,T2,T3):-  reduct5(T1,R1), T1 \== R1,
        lowunifyd5(R1,T2,T3).
lowunifyd5(T1,T2,T3):-  reduct5(T2,R2), T2 \== R2,
        lowunifyd5(T1,R2,T3).
lowunifyd5(T1,T2,T3):-  reduct5(T1,R1), reduct5(T2,R2),
        T1 \== R1, T2 \== R2,
        lowunifyd5(R1,R2,T3).

lowunifyd45(T1,T2,T3):- arg(1,T1,w(Y)),  arg(1,T2,w(X)),
        X \== Y, !, fail.
lowunifyd45(T1,T2,T3):- lowunifyd5(T1,T2,T3).
lowunifyd45(T1,T2,T3):-  reduct4(T1,R1), T1 \== R1,
        lowunifyd45(R1,T2,T3).
lowunifyd45(T1,T2,T3):-  reduct4(T2,R2), T2 \== R2,
        lowunifyd45(T1,R2,T3).
lowunifyd45(T1,T2,T3):-  reduct4(T1,R1),
        reduct4(T2,R2), T1 \== R1, T2 \== R2,
        lowunifyd45(R1,R2,T3).

/* high unification */

unifyk(T1,T2,T3):- arg(1,T1,H1), arg(1,T2,H2),
        (H1 = w(N); H2 = w(N)),
        unifyd(T1,T2,T3).

unifyd4(T1,T2,T3):- mycompare(>,T1,T2),
        arg(1,T2,vw(N)), arg(2,T1,B1),
        arg(1,T1,H1), unifyd(H1,vw(N),H3),
        functor(T3,i,2), arg(1,T3,H3), arg(2,T3,B3),
        (unifyd(B1,T2,B3); unifyd4(B1,T2,B3)).

unifyd4(T1,T2,T3):- mycompare(<,T1,T2),
        arg(1,T1,vw(N)), arg(2,T2,B2),
        arg(1,T2,H2), unifyd(H2,vw(N),H3),
        functor(T3,i,2), arg(1,T3,H3), arg(2,T3,B3),
        (unifyd(T1,B2,B3); unifyd4(T1,B2,B3)).

unifyd5(T1,T2,T3):- (arg(1,T1,w(N));
        mycompare(=,T1,T2)), arg(1,T2,H2),
```

```
        unifyd(w(N),H2,H3),
        arg(2,T1,B1),  arg(2,B1,BB1), arg(2,T2,B2),
        (unifyd(BB1,B2,B3); unifyd5(BB1,B2,B3)),
        functor(T3,i,2), arg(1,T3,H3), arg(2,T3,B3).

unifyd5(T1,T2,T3):- (arg(1,T2,w(N));
        mycompare(=,T1,T2)) , arg(1,T1,H1),
        unifyd(H1,w(N),H3),
        arg(2,T2,B2),  arg(2,B2,BB2), arg(2,T1,B1),
        (unifyd(B1,BB2,B3); unifyd5(B1,BB2,B3)),
        functor(T3,i,2), arg(1,T3,H3), arg(2,T3,B3).

unifyd5(T1,T2,T3):- (arg(1,T1,w(N));
        mycompare(>,T1,T2)) , arg(1,T2,H2),
        unifyd(w(N),H2,H3),
        arg(2,T1,B1),  arg(2,B1,BB1), arg(2,T2,B2),
        unifyd5(BB1,B2,B3),
        functor(T3,i,2), arg(1,T3,H3), arg(2,T3,B3).

unifyd5(T1,T2,T3):- (arg(1,T2,w(N));
        mycompare(<,T1,T2)) , arg(1,T1,H1),
        unifyd(H1,w(N),H3),
        arg(2,T2,B2),  arg(2,B2,BB2), arg(2,T1,B1),
        unifyd5(B1,BB2,B3),
        functor(T3,i,2), arg(1,T3,H3), arg(2,T3,B3).

mycompare(Rel,T1,T2):- term_depth(T1,N1),
        term_depth(T2,N2), compare(Rel,N1,N2).
```

Example

```
| ?- pr(d,[~((@a + $b) -> $(a+b))]).
[i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]<--->[]
alpha elimination
[i(w(1),w(1)): @a+ $b,i(w(1),w(1)): ~ $ (a+b)]<--->[i(w(1),w(1)): ~ (@a+ $b-> $(a+b))]
pi elimination
[i(w(1),w(1)): @a+ $b,i(w(2),i(w(1),w(1))): ~ (a+b)]<--->[i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
alpha elimination
[i(w(1),w(1)): @a+ $b,i(w(2),i(w(1),w(1))): ~a,i(w(2),i(w(1),w(1))): ~b]<--->[i(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),
i(w(1),w(1)): ~ (@a+ $b-> $(a+b))]
literal
[i(w(1),w(1)): @a+ $b,i(w(2),i(w(1),w(1))): ~b]<--->[i(w(2),i(w(1),w(1))): ~a,i(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),
i(w(1),w(1)): ~ (@a+ $b-> $(a+b))]
literal
i(w(1),w(1)): @a+ $b, i(w(1),w(1)): @a unify in  d
pb1
branch 1
[i(w(1),w(1)): @a,i(w(1),w(1)): @a+ $b]<--->[i(w(2),i(w(1),w(1))): ~b,i(w(2),i(w(1),w(1))): ~a,i(w(2),i(w(1),w(1))): ~ (a+b),
i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
pi elimination
[i(w(3),i(w(1),w(1))):a,i(w(1),w(1)): @a+ $b]<--->[i(w(1),w(1)): @a,i(w(2),i(w(1),w(1))): ~b,i(w(2),i(w(1),w(1))): ~a
,i(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
literal
i(w(1),w(1)): @a+ $b, i(w(1),w(1)): @a unify in  d
[i(w(1),w(1)): @a+ $b]<--->[i(w(3),i(w(1),w(1))):a,i(w(1),w(1)): @a,i(w(2),i(w(1),w(1))): ~b,i(w(2),i(w(1),w(1))): ~a
,i(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
beta semplification 1
[i(w(1),w(1)): @a+ $b,i(w(3),i(w(1),w(1))):a,i(w(1),w(1)): @a,i(w(2),i(w(1),w(1))): ~b,i(w(2),i(w(1),w(1))): ~a,
(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $(a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]***completed
branch 2
[i(w(1),w(1)): ~ @a,i(w(1),w(1)): @a+ $b]<--->[i(w(2),i(w(1),w(1))): ~b,i(w(2),i(w(1),w(1))): ~a,i(w(2),i(w(1),w(1))): ~ (a+b),
(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
ni elimination
[i(vw(1),i(w(1),w(1))): ~a,i(w(1),w(1)): @a+ $b]<--->[i(w(1),w(1)): ~ @a,i(w(2),i(w(1),w(1))): ~b,i(w(2),i(w(1),w(1))): ~a,
(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
literal
i(w(1),w(1)): @a+ $b, i(w(1),w(1)): ~ @a unify in  d
[i(w(1),w(1)): @a+ $b]<--->[i(vw(1),i(w(1),w(1))): ~a,i(w(1),w(1)): ~ @a,i(w(2),i(w(1),w(1))): ~b,i(w(2),i(w(1),w(1))): ~a,
(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
beta elimination 1
```

[i(w(1),w(1)): $b]<--->[@a+ $b,i(vw(1),i(w(1),w(1))): ~a,i(w(1),w(1)): ~ @a,i(w(2),i(w(1),w(1))): ~b,i(w(2),i(w(1),w(1))): ~a,(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
ni elimination
[i(vw(1),i(w(1),w(1))):b]<--->[i(w(1),w(1)): $b,@a+ $b,i(vw(1),i(w(1),w(1))): ~a,i(w(1),w(1)): ~ @a,i(w(2),i(w(1),w(1))): ~b,(w(2),i(w(1),w(1))): ~a,i(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))]
literal
i(w(2),i(w(1),w(1))): ~b, i(vw(1),i(w(1),w(1))):b unify in d
[i(vw(1),i(w(1),w(1))):b,i(w(1),w(1)): $b,@a+ $b,i(vw(1),i(w(1),w(1))): ~a,i(w(1),w(1)): ~ @a,i(w(2),i(w(1),w(1))): ~b,(w(2),i(w(1),w(1))): ~a,i(w(2),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ $ (a+b),i(w(1),w(1)): ~ (@a+ $b-> $ (a+b))] ***closed
satisfiable in d
in 267 msecs.

References

[ACG94a] A. Artosi, P. Cattabriga and G. Governatori. *KEM*: A Modal Theorem Prover. Forthcoming in *Annali dell'Università di Ferrara*, Sez. III, Fil., Discussion paper series.

[ACG94b] A. Artosi, P. Cattabriga and G. Governatori. *An Automated Approach to Deontic Reasoning.* Paper submitted to the ECAI Workshop on Artificial Normative Reasoning.

[AE92] Y. Auffray, and P. Enjalbert. Modal Theorem Proving: An Equational Wiewpoint. *Journal of Logic and Computation*, 2, 1992: 247-259.

[AEH90] Y. Auffray, P. Enjalbert and J-J. Herbrard. Strategies for Modal Resolution: Results and Problems. *Journal of Automated Reasoning*, 6, 1990: 1-38.

[AG93] A. Artosi and G. Governatori. Labelled Modal Proofs. In A. Artosi (ed.), Two Papers in Proof Theory. *Annali dell'Università di Ferrara*, Sez. III, Fil., Discussion paper series 33, 1993.

[AG94] A. Artosi and G. Governatori. Labelled Model Modal Logic. Forthcoming in *Proceedings of The CADE 12 Workshop on Automated Model Building.*

[Åq87] L. Aqvist. *Introduction to Deontic Logic and the Theory of Normative Systems*, Bibliopolis, Napoli, 1987.

[Bel87] M. Belzer. Legal Reasoning in 3-D. In *Proceedings of the First International Conference on Artificial Intelligence and Law*, ACM Press, New York, 1987: 155-63.

[Boo79] G. Boolos. *The Unprovability of Consistency*. Cambridge University Press, Cambridge, 1979.

[Cat91] L. Catach. Tableaux: A General Theorem Prover for Modal Logics. *Journal of Automated Reasoning*, 7, 1991: 489-510.

[Ch80] B. F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, Cambridge, 1980.

[Cha87] M. Chan. The Recursive Resolution Method for Modal Logic. *New Generation Computing*, 5, 1987: 155-183.

[Cia86] M. Cialdea. Some Remarks on the Possibility of Extending Resolution Proof Procedures to Intuitionistic Logic. *Information Processing Letters*, 22, 1986: 87-90.

[DG93] M. D'Agostino and D.M. Gabbay. Labelled Refutation Systems: A Case-study. Draft Manuscript, Imperial College, March 1993.

[DM94] M. D'Agostino and M. Mondadori. The Taming of the Cut. Forthcoming in *Journal of Logic and Computation*, 4, 1994.

[DP94] M. D'Agostino and J. Pitt. Private comunication.

[EF89] P. Enjalbert and L. Fariñas del Cerro. Modal Resolution in Clausal Form. *Theoretical Computer Science*, 1, 1989: 1-33.

[Far85] L. Fariñas del Cerro. Resolution Modal Logic. *Logique et Analyse*, 110-111, 1985: 152-172.

[Far86] L. Fariñas del Cerro. Molog: A System That Extends Prolog with Modal Logic. *New Generation Computing*, 4, 1986: 35-50.

[FH88] L. Fariñas del Cerro and Andreas Herzig. Linear Modal Deductions. *Lecture Notes in Computer Science*, 310, Springer-Verlag, 1988: 487-499.

[Fi66] F. B. Fitch, Tree Proofs in Modal Logic (abstract). *Journal of Symbolic Logic*, 31, 1966: 152.

[Fit72] M. Fitting, Tableau Methods of Proof for Modal Logic. *Notre Dame Journal of Formal Logic*, 13, 1972: 237-247.

[Fit83] M. Fitting. *Proof Methods for Modal and Intuitionistic Logic*, Reidel, Dordrecht, 1983.

[Fit88] M. Fitting. First-Order Modal Tableaux. *Journal of Automated Reasoning*, 4, 1988: 191-213.

[GD88] C. Groeneboer and J. Delgrande. Tableau-Based Theorem Proving in Normal Conditional Logics. In :*Proceedings AAAI '88*: 171-176.

[Gen93] I. P. Gent. Theory Matrices (for Modal Logics) Using Alphabetical Monotonicity. *Studia Logica*, 52.1993: 233-257.

[Han65] W. H. Hanson. Semantics for Deontic Logic. *Logique et Analyse*, 31, 1965: 177-190.

[JP85] A. J. I. Jones and I. Pörn. Idality, Sub-Ideality and Deontic Logic. *Synthese*, 65, 1985: 275-290.

[JR89] P. Jackson and H. Reichgelt. A General Proof Method for Modal Predicate Logic. In P. Jackson, H. Reichgelt, and F. van Harmelen. *Logic-Based Knowledge Representation*. The MIT Press, Cambridge, Mass., 1989.

[McC83] L.T. McCarty. Permissions and Obligations. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, Karlsruhe, 1983: 287-294.

[McC86] L.T. McCarty. Permissions and Obligations: An Informal Introduction. In A.A. Martino and F. Socci Natali (eds.), *Automated Analysis of Legal Texts: Logic, Informatics, Law*. North-Holland, Amsterdam, 1986: 307-337.

[Mon88] M. Mondadori. Classical Analytical Deduction. *Annali dell'Università di Ferrara*, Sez. III, Fil., Discussion paper series 1, 1988.

[Ohl87] H. J. Ohlbach. A Resolution Calculus for Modal Logics. *Lecture Notes in Computer Science*, 310, Springer-Verlag, 1987: 500-516.

[Ohl89] H. J. Ohlbach. New Ways for Developing Proof Theories for First-Order Multi Modal Logics. *Lecture Notes in Computer Science*, Springer-Verlag, 1989:271-308.

[Ohl91] H. J. Ohlbach. Semantics-Based translation Methods for Modal Logics. *Journal of Logic and Computation* 1, 1991: 691-746.

[Ohl93] H. J. Ohlbach. Translation Methods for Non -Classical Logics: An Overview. *Bulletin of the IGPL* 1, 1993: 69-89.

[Sus87] R.E. Susskind. *Expert Systems in Law: A Jurisprudential Inquiry*. Clarendon Press, Oxford, 1987.

[Tap87] B. L. Tapscott. A Simplified Natural Deduction Approach to Certain Modal Systems. *Notre Dame Journal of Formal Logic*, 28, 1987: 371-383.

[Wal90] L. A. Wallen. *Automated Deduction in Non-Classical Logics*, The MIT Press, Cambridge, Mass., 1990.

[Wol85] P. Wolper. The Tableau Method for Temporal Logic: an Overview. *Logique et Analyse*, 110-111, 1985: 119-136.

[Wri85] G. Wrightson. Non-Classical Logic Theorem Proving, *Journal of Automated Reasoning*, 1, 1985: 35-37.