An Automated Approach to Deontic Reasoning

Alberto Artosi[*], *Paola Cattabriga*[**], *Guido Governatori*[**]
[*] *Dipartimento di Filosofia, Università di Bologna,*
via Zamboni,38, 40126 Bologna (Italy), Fax:+39-51-258326
[**] *CIRFID, Università di Bologna,*
via Galliera, 3, 40126 Bologna (Italy), Fax: +39-51-260782
E-Mail: governat@cirfid.unibo.it, paola@cirfid.unibo.it

1 Introduction

It is by now generally accepted in the Artificial Intelligence and Law field that many aspects of normative language and reasoning can be modelled in deontic logics based on modal logic (see e.g. [Jon90], [JS91]). This obviously implies the need for computationally tractable inference mechanisms for Deontic Logic (see e.g. [Bel87], [McC83], [McC86]). In this paper we shall be concerned with developing a computationally oriented proof method for several normal (in [Åqv84] classification normal and strongly normal) deontic logics. Since this method is arguably more natural and intuitive than other (e.g. resolution or translation based) proof methods, and it leads to simple and easy implementable procedures, it seems particularly well-suited for applications in the newly developed area of "artificial normative reasoning". Moreover, though in the present version it works for deontic logics of the simplest kind, it is sufficiently generic and flexible to provide an appropriate algorithmic proof framework for deontic logics of greater richness and complexity. The paper is organized as follows. Section 2 provides a short description of the systems of deontic logic we will deal with together with an explanation of the logical notation being used. Section 3 describes the deontic theorem prover KED. Section 4 presents KED method of proof search, while Section 5 provides concluding remarks. Finally, the last Section provides (a sample of) the Prolog implementation of the method and gives an example output of the program.

2 Preliminaries

All the systems of Deontic Logic we shall be concerned with are couched in a standard modal language consisting of a denumerable set of propositional variables and the primitive logical connectives $\neg, \wedge, \vee, \rightarrow \mathsf{P}, \mathsf{O}$ for negation, conjunction, disjunction, conditionality,

permission and obligation, respectively. We shall use the letters $A, B, C, \ldots$ to denote arbitrary formulas of this language. A system of Deontic Logic will be denoted by L. We define a *Kripke model* for a logic L (briefly an *L-model*) to be a triple $\langle \mathcal{W}, \mathcal{R}, \upsilon \rangle$ where $\mathcal{W}$ is a non-empty set (the set of "possible worlds"), $\mathcal{R}$ is a binary relation on $\mathcal{W}$ (the "accessibility relation" between the "actual" world and its deontically ideal versions), and υ is a mapping from $\mathcal{W} \times \mathcal{S}$ to $\{T, F\}$ where $\mathcal{S}$ 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 $\mathcal{R}$ to satisfy the conditions associated with L. The following table summarizes the logics to be dealt with.

L	**Definition**	**Condition on $\mathcal{R}$**
OK	$PC + \mathsf{O}(A \to B) \to (\mathsf{O}A \to \mathsf{O}B)$	no condition
OM	$OK + \mathsf{O}(\mathsf{O}A \to A)$	almost reflexive
$OS4$	$OM + \mathsf{O}A \to \mathsf{OO}A$	transitive, almost reflexive
OUB	$OM + \mathsf{O}(A \to \mathsf{OP}A)$	almost symmetric, almost reflexive
$OS5$	$OS4 + \mathsf{P}A \to \mathsf{OP}A$	euclidean, transitive
D	$OK + \mathsf{O}A \to \mathsf{P}A$	idealisation
DM	$OM + \mathsf{O}A \to \mathsf{P}A$	idealisation, almost reflexive
$DS4$	$OS4 + \mathsf{O}A \to \mathsf{P}A$	idealisation, transitive, almost reflexive
DUB	$OUB + \mathsf{O}A \to \mathsf{P}A$	idealisation,almost symmetric, almost reflexive
$DS5$	$OS5 + \mathsf{O}A \to \mathsf{P}A$	idealisation, euclidean, transitive
OB	$OK + A \to \mathsf{OP}A$	symmetric
$O5$	$OK + \mathsf{P}A \to \mathsf{OP}A$	euclidean
OMB	$OM + A \to \mathsf{OP}A$	symmetric, almost reflexive
DB	$D + A \to \mathsf{OP}A$	idealisation, symmetric
$D4$	$DM + \mathsf{O}A \to \mathsf{OO}A$	idealisation, transitive
$D5$	$D + \mathsf{P}A \to \mathsf{OP}A$	idealisation, euclidean

Table 1: Systems of Deontic Logic 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 $\mathsf{O}A$). The condition associated with the D-logics corresponds to the obvious requirement that every world in $\mathcal{W}$ has at least an ideal version.

Remark 1: Of the logics just listed, $OK, OM, OS4, OUB, OS5$ are identical, respectively, with the normal systems $OK, OM, OS4, OB, OS5$ of [Åqv84]. $D, DM, DS4, DUB, DS5$ are identical, respectively, with [Åqv84] strongly normal systems OK^+ ([Han65] D and [Ch80] D^*), $OM^+, OS4^+, OB^+$ ([Han65] DB), $OS5^+$. OMB is considered neither by [Åqv84] nor by [Han65]. $OB, O5, DB, D4, D5$ are (in Chellas [Ch80] classification) normal KB-, $K5$- and KD-systems

As usual [Smu68] by a *signed formula (S-formula)* we shall mean an expression of the form SA where A is a formula and $S \in \{T, F\}$. Thus TA if $(\upsilon, A) = T$ and FA if $(\upsilon, A) = F$ for some L-model $\langle \mathcal{W}, \mathcal{R}, \upsilon \rangle$ and $x \in \mathcal{W}$. 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 TA is the conjugate of FA and FA is the conjugate of TA). 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
$TA \wedge B$	TA	TB	$FA \wedge B$	FA	FB
$FA \vee B$	FA	FB	$TA \vee B$	TA	TB
$FA \rightarrow B$	TA	FB	$TA \rightarrow B$	FA	TB
$T\neg A$	FA	FA	$F\neg A$	TA	TA

ν	ν_0	π	π_0
$T\mathsf{O}A$	TA	$T\mathsf{P}A$	TA
$F\mathsf{P}A$	FA	$F\mathsf{O}A$	FA

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. Like resolution and tableau systems, KED is a formalization of the search for countermodels. The key feature of KED, besides its being based neither on variants of the resolution method nor on the standard "cut free" tableau calculus but on a combination of tableau and natural deduction inference rules which allows for a suitably restricted use of the cut rule, is that it automatically generates models and checks them for putative contradictions using a label scheme to bookkeep "world" paths. Briefly and informally, in the KED-based approach S-formulas are labelled by worlds. A "world" label is a constant or a variable "world" symbol or a "structured" sequence of world-symbols we shall call a "world-path". Intuitively, constant and variable world-symbols can be viewed as denoting worlds and sets of worlds respectively, while a world-path conveys information about access between the worlds in it. An S-formula SA with an associated label i (a *labelled signed formula*, or LS-formula, as we shall call it) means, intuitively, that A is true (false) at the (last) world (on the path) i. In the course of proof search, labels are manipulated in a way closely related to the semantics of deontic operators and "matched" using a (specialized, logic-dependent) unification algorithm. That two structured labels i and k are unifiable means, intuitively, that they virtually represent the same path, i.e. any world which you could get to by the path i could be reached by the path k and vice versa. S-formulas whose labels are unifiable turn out to be true (false) at the same world(s) relative to the accessibility restrictions that hold in the class of L-models. In particular, two LS-formulas $X, i\ X^C, k$ whose labels are unifiable will stand for formulas which are contradictory "in the same world".

Remark 2: The idea of using a label scheme to bookkeep "world" paths in modal theorem proving goes back at least to [Fi66]. Similar, or related, ideas are found in [Fit72], [Fit83] and [Wri85] and, more recently, in [Cat91], [JR89], [Tap87], [Wal90] and also in the "translation" tradition of [AE92], [Ohl93], and in Gabbay's Discipline of Labelled Deductive Systems (see [DG93] tableau extension with labels).

3.1 Label formalism

A world-label is defined to be either (i) an element of a (non empty) set $\Phi_C = \{w_1, w_2, w_3, \ldots\}$ of constant world-symbols, or (ii) an element of a (non empty) set $\Phi_V = \{W_1, W_2, W_3, \ldots\}$ of world-variables, or (iii) a path term (k', k) where (iiia) $k' \in \Phi_C \cup \Phi_V$ and (iiib) $k \in \Phi_C$

or $k = (m', m)$ where (m', m) is a label from now on we shall use $\Im$ to denote the set of world-labels. According to the above intuitive explanation, 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 the set W_1 of worlds accessible from (i.e. which are the deontically ideal version of the "real 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 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 define the length of i, $l(i)$, as the number of world-symbols in i. We call a label i restricted if $h(i) \in \Phi_C$, otherwise we call it unrestricted.

3.2 Basic unifications

We define a substitution in the usual way as a function $\sigma : \Phi_V \to \Im^-$ where $\Im^- = \Im - \Phi_V$. For two labels $i,\ k$ and a substitution σ we shall use $(i, k)\sigma$ to denote both that i and k are unifiable (briefly, are σ-unifiable) and the result of their unification. On this basis we define several specialised, logic-dependent notions of σ-unification. We define the notion of two labels i, k being σ^L-unifiable in the following way:

$(i, k)\sigma^O = (i, k)\sigma \iff$
(i) at least one of i and k is restricted, and
(ii) for every $s(i),\ s(k),\ l(s(i)) = l(s(k)),\ (s(i), s(k))\sigma^O$

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

$(i, k)\sigma^{XM} = (h(s(i)), h(k))\sigma \times ((h(s^m(i)), h(s^m(k)))\sigma \times (\ldots \times (s^2(i), s^2(k)\sigma^X))) \iff$
$2 \leq l(i) < l(k) \forall h(s(i)) : l(s(i)) \geq l(k),\ (h(s(i)), h(k))\sigma^X = (h(i), h(k))\sigma^X$ or

$(i, k)\sigma^{XM} = (h(s(i)), h(k))\sigma \times ((h(s^m(i)), h(s^m(k)))\sigma \times (\ldots \times (s^2(i), s^2(k)\sigma^X))) \iff$
$2 \leq l(i) < l(k) \forall h(s(i)) : l(s(i)) \geq l(k),\ (h(s(i)), h(k))\sigma^X = (h(i), h(k))\sigma^X$

where $2 \leq m < l(i), (l(k))$; $s^t(i)$ denotes the segment of i on any lenght t, and $X = O; D$ for the O- D- logics respectively.

$(i, k)\sigma^{X4} = h(k) \times h(b(k)) \times (\ldots \times (t^*(k) \times (i, s(k))\sigma^X)\ldots)) \iff$
$l(i) \leq l(k)$ and $(i, s(k))\sigma^X,\ h(i) \in \Phi_V$, or

$(i, k)\sigma^{X4} = h(i) \times h(b(i)) \times (\ldots \times (t^*(i) \times (s(i), sk)\sigma^X)\ldots)) \iff$
$l(k) \leq l(i)$ and $(s(i), k)\sigma^X,\ h(k) \in \Phi_V$

where $t^*(k)$ (resp. $t^*(i)$) denotes the element of k (resp. i) which immediately follows $s(k)$ (resp. $s(i)$) and $X = O, D$, for the O- and D-logics respectively

$$(i, k)\sigma^{OS4} = \begin{cases} (i, k)\sigma^{OM} & h(shortest\{i, k\} \in \Phi_C \\ (i, k)\sigma^{O4} & h(shortest\{i, k\} \in \Phi_V \end{cases}$$

$$(i,k)\sigma^{DS4} = \begin{cases} (i,k)\sigma^{DM} & h(shortest\{i,k\} \in \Phi_C \\ (i,k)\sigma^{D4} & h(shortest\{i,k\} \in \Phi_V \end{cases}$$

$(i,k)\sigma^{X5} = (h(i),h(k))\sigma^X \times (b(b(i)),b(k))\sigma^L \iff$
$(h(i),h(k))\sigma$ and $(b(b(i),b(k))\sigma^L$ if $h(i) \in \Phi_V$, or

$(i,k)\sigma^{X5} = (h(i),h(k))\sigma \times (b(i),b(b(k)))\sigma^L \iff$
$(h(i),h(k))\sigma$ and $(b(i),b(b(k)))\sigma^L$ if $h(k) \in \Phi_V$

where

$$\sigma^L = \begin{cases} \sigma^X \text{ or } \sigma^{X5} & \text{if } l(i) = l(k) \\ \sigma^{X5} & \text{if } l(i) \neq l(k) \end{cases}$$

for $X = O, D$ and, if $X = O$, at least one of $h(i), h(k), b(i), b(k)$ is in Φ_C.

$(i,k)\sigma^{OS5} = (h(i),h(k))\sigma^O \times (s^1(i),s^1(k))\sigma^O$
if $l(i), l(k) > 1, (s^2(i),s^2(k))\sigma^O$

$(i,k)\sigma^{DS5} = (h(i),h(k))\sigma^D \times (s^1(i),s^1(k))\sigma^D$
if $l(i), l(k) > 1$

The above notions are meant to mirror the conditions on $\mathcal{R}$ in the various L-models. For example, the crucial notions of σ^O-and σ^D-unification are related to the idealisation condition; thus $(w_2,(W_1,w_1)),(W_3,(W_2,w_1))$ are σ^D-unifiable but not σ^O-unifiable (since the segments $(W_1,w_1),(W_2,w_1)$ are not σ^O-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), which obviously makes their unification impossible, while in the "idealisable" logic D they are not empty, which makes them unifiable "on" any constant.

3.3 Reductions

For $X = 4, B, UB, 5$ we define the *X-reduction*, $r_X(i)$, of a label i to be a function $r_X : \Im \to \Im$ determined as follows:

$$r_4(i) = \begin{cases} (h(i),b(b(i))) & i \text{ restricted and } l(i) > 3 \\ (h(i),r_4(b(i))), & i \text{ unrestricted} \end{cases}$$

$$r_B(i) = \begin{cases} b(b(i)), & i \text{ unrestricted and } l(i)) \leq 3 \text{ or } b(i) \text{ restricted} \\ (h(i),r_B(b(i))), & \text{otherwise} \end{cases}$$

$$r_{UB}(i) = \begin{cases} b(b(i)), & i \text{ unrestricted and } l(i) \leq 4 \text{ or } b(i) \text{ restricted} \\ (h(i),r_{UB}(b(i))), & \text{otherwise} \end{cases}$$

$$r_5(i) = \begin{cases} (h(i),b(b(i)) & \text{if } i, b(i) \text{ unrestricted} \\ (h(i),r_5 b(i)) & \text{otherwise} \end{cases}$$

The notion of X-reduction holds for all transitive, symmetric, almost symmetric, and euclidean logics. 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 $\mathcal{R}$ is constrained to satisfy 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

$L =$

$OK:\ (i,k)\sigma_{OK} = (i,k)\sigma^{O}.$

$OM:\ (i,k)\sigma_{OM} = \begin{cases} (i,k)\sigma^{O} & l(i) = l(k) \\ (i,k)\sigma^{M} & l(i) \neq l(k) \end{cases}$

$OS4:\ (i,k)\sigma_{OS4} = \begin{cases} (r_4(i,k))\sigma^{O} & l(i) = l(k) \\ (r_4(i,k))\sigma^{OS4} & l(i) \neq l(k) \end{cases}$

$OUB:\ (i,k)\sigma_{OUB} = (r_{UB}(i,k))\sigma_{OK}$

$OS5:\ (i,k)\sigma_{OS5} = (i,k))\sigma^{OS5}$

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

$DM:\ (i,k)\sigma_{DM} = \begin{cases} (i,k)\sigma^{D} & l(i) = l(k) \\ (i,k)\sigma^{M} & l(i) \neq l(k) \end{cases}$

$DS4:\ (i,k)\sigma_{DS4} = \begin{cases} (r_4(i,k))\sigma^{O} & l(i) = l(k) \\ (r_4(i,k))\sigma^{DS4} & l(i) \neq l(k) \end{cases}$

$DUB:\ (i,k)\sigma_{DUB} = (r_{UB}(i,k))\sigma_{D}$

$DS5:\ (i,k)\sigma_{DS5} = (i,k)\sigma^{DS5}$

$DMB:\ (i,k)\sigma_{DMB} = (r_{UB}(i,k))\sigma_{DM}$

$D4:\ (i,k)\sigma_{D4} = \begin{cases} (r_4(i,k))\sigma^{D} & l(i) = l(k) \\ (r_4(i,k))\sigma^{D4} & l(i) \neq l(k) \end{cases}$

$D5:\ (i,k)\sigma_{D5} = \begin{cases} (r_5(i,k))\sigma^{O} & l(i) = l(k) \\ (r_4(i,k))\sigma^{D5} & l(i) \neq l(k) \end{cases}$

$OMB:\ (i,k)\sigma_{OMB} = (r_{UB}(i,k))\sigma_{OM}$

$O5:\ (i,k)\sigma_{O5} = \begin{cases} (r_5(i,k))\sigma^{O} & l(i) = l(k) \\ (r_5(i,k))\sigma^{O5} & l(i) \neq l(k) \end{cases}$

where $r_X(i,k)$ denotes either $r_X(i)$ or $r_X(k)$ or both.

3.4 Rules of inference

The rules of KED will be defined for LS-formulas. Two LS-formulas $X, i,\ Z, k$ such that $Z = X^C$ and $(i,k)\sigma_L$ will be called $\sigma_L - complementary$. The following inference rules hold for all the logics we are considering (i, i', k stand for arbitrary labels).

$$\frac{\alpha, i}{\alpha_1, i} \qquad \frac{\alpha, i}{\alpha_2, i}$$

$$\frac{\begin{matrix}\beta, i \\ \beta_1^C, k\end{matrix}}{\beta_2, (i,k)\sigma_L}[(i,k)\sigma_L] \qquad \frac{\begin{matrix}\beta, i \\ \beta_2^C, k\end{matrix}}{\beta_1, (i,k)\sigma_L}[(i,k)\sigma_L]$$

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

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

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. PB (the "Principle of Bivalence") 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). PNC (the "Principle of Non-Contradiction") corresponds to the familiar branch-closure rule of the tableau method, saying that from a contradiction of the form (the occurrence of a pair of σ_L-complementary LS-formulas) $X, i,\ X^C, k$ on a branch we may infer the closure of the branch. The $(i, k)\sigma_L$ in the "conclusion" of PNC means that the contradiction holds "in the same world".

3.5 Soundness and completeness

Soundness and completeness of KED derive from the following

THEOREM 1 $\vdash_L A \Leftrightarrow \vdash_{KED(L)} A$ for each L

where as usual we write $\vdash_L A$ to mean that there is a proof of A in the axiomatic L and $\vdash_{KED(L)} A$ is used to denote that there is a proof of A in the KED version of L. For the proof of this theorem see [AG93].

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 the last section). In what follows by a *KED-tree* we shall 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 $\mathcal{T}$ will be said to be *σ_L-closed* if all its branches are σ_L-closed. Finally, by a L-proof of a formula A we shall mean σ_L-closed KED-tree starting with FA, i. Given a branch τ of 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 β_1^C, k occurs in τ and $(i, k)\sigma_L$, then also $\beta_2, (i, k)\sigma_L$ occurs in τ, (b) if β_2^C, k occurs in τ and $(i, k)\sigma_L$, then also $\beta_1, (i, k)\sigma_L$ occurs in τ; or (iii) X is of type ν and $\nu_0, (i', i)$ occurs in τ for some $i' \in \Phi_V$ not previously occurring in τ, or (iv) X is of type π and $\pi_0, (i', i)$ occurs in τ for some $i' \in \Phi_C$ 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 either are fulfilled or cannot be fulfilled.. We shall call a KED-tree *completed* if every branch is completed. Let us denote by Δ (delta) the set of LS-formulas which occur non analysed, by Λ (lambda) the set of analysed LS-formulas, and by $\mathcal{L}$ the set of labels. The KED algorithm runs as follows (the quotations in brackets refer to the Prolog implementation in the section 5). To prove a formula A of L

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

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

STEP 2 (cke 3). Every literal is deleted from Δ, and added to Λ. 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 $\mathcal{L}$; (ii) delete π,i (ν,i) from Δ; (iii) add π_0,(i',i) (ν_0,(i',i)) to Δ; and (iv) add π,i (ν,i) to Λ.

STEP 5 (cke 7). For each formula α,i in Δ, (i) add α_1,i, and 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) σ_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 β_1^C,k orβ_2^C,k is in $\Delta \cup \Lambda$ and (i,k) σ_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; and (iii) add the labels resulting from the σ_L-unification to $\mathcal{L}$; 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 contains formulas β_1^C,k such that i, k are not σ_L-unifiable, then form sets $\Delta_1 = \Delta \cup \beta_1, m$ and $\Delta_2 = \Delta \cup \beta_1^C, m$ where $(i,m)\sigma_L$, and m is a given restricted label.

STEP 8.2 (cke 12). For each formula β,i in Δ, if $\Delta \cup \Lambda$ does not contains formulas β_2^C,k so that i, k are not σ_L-unifiable, then form sets $\Delta_1 = \Delta \cup \beta_2$,$m$ and $\Delta_2 = \Delta \cup \beta_2^C$,$m$ where $(i$,$m)\sigma_L$, and m is a given restricted label.

Remark 3:The steps 8.1 and 8.2 are logic and label dependent. This mean that if the label of X is restricted, its immediate subformulas have the same label as X, otherwise we have to deal with two cases: a) search whether $\mathcal{L}$ 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 $\mathcal{L}$ 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. A is a theorem of L.

STEP 10 (cke15). If Λ contains two complementary but not σ_L-complementary formulas, search in $\mathcal{L}$ 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 canon-

ical iff all the applications of 1-premise rule come before the applications of 2-premise rules, which preceed the applications of the 0-premise rule. Some interessing property of canonical KED-trees are stated in the followings

THEOREM 2. A canonical KED-tree always terminates.

THEOREM 3. A KED-tree for a formula A is closed iff the canonical KED-tree for A is closed.

The proof of theorem 2 derives 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 belonging to A. The proof of theorem 3 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 [ACG94b]).

Remark 4:The above procedure closely follows the canonical restriction of Mondadori's ([Mon88]) classical proof system KE in applying the cut rule PB *only* to immediate signed subformulas of LS-formulas of type which occur (unfulfilled) in the chosen branch, and *only* when the branch has been E-completed. As D'Agostino and Mondadori ([DM94]) proved, the canonical restriction of KE retains all the desirable properties of the analytic tableaux (subformula property and easy proof search), while uniformly and essentially improving on them from the point of view of computational efficiency.

5 Final remarks

In our opinion, the interest in the system thus presented is that it offers several advantages over most automated theorem proving systems for non-classical logics currently available. Here we mention only a few:

- it requires no preprocessing of the input formulas (i.e., no conversion to any normal form or translation procedure);
- it provides a simple and uniform treatment of a wide variety of normal modal logics (see [AG94]);
- it supports a deduction method closely related to the semantics of modal operators;
- it directly implements familiar inference patterns and yields proofs in a good natural inference style;
- it avoids loop-checking and reduplication (see Section 4 above);
- it works for modal logics whose characteristic semantic properties have no first-order characterization (such as the Gödel-Löb logic of provability [Bo79]).

Furthermore, as said before it provides an appropriate algorithmic proof framework for more expressive and sophisticated deontic logics which have become of some importance in the Artificial Intelligence and Law field, such as deontic logics of conditional obligation or Jones and Pörn [JP85] system DL (as indicated in [ACG94a]).

6 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 L. If the input formula is a theorem of L the program will output the entire reserch path with the final answer "closed", 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 [ACG94b]). 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 the program `->, +, &, ~ , $, @` stand respectively for $\rightarrow, \vee, \wedge, \neg, \mathsf{O}, \mathsf{P}$.

Example

We show the Prolog output of the D-theorem

$$(\mathsf{P}A \vee \mathsf{O}B) \rightarrow \mathsf{P}(A \vee B)$$

```
| ?- pr(d,[~((@a + $b) -> @(a+b))]).
output
[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))]
ni elimination
[i(w(1),w(1)): @a+ $b,i(vw(1),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(vw(1),i(w(1),w(1))): ~a,i(vw(1),i(w(1),w(1))): ~b]<--->[i(vw(1),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(vw(1),i(w(1),w(1))): ~b]<---> [i(vw(1),i(w(1),w(1))): ~a,i(vw(1),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(vw(1),i(w(1),w(1))): ~b,i(vw(1), i(w(1),w(1))): ~a,i(vw(1),
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(2),i(w(1),w(1))):a, i(w(1),w(1)): @a+ $b]<--->[i(w(1),w(1)): @a,i(vw(1),i(w(1),w(1))): ~b,i(vw(1),
i(w(1),w(1))): ~a,i(vw(1),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(vw(1),i(w(1),w(1))): ~a, i(w(2),i(w(1),w(1))):a unify in  d
[i(w(2),i(w(1),w(1))):a,i(w(1),w(1)): @a,i(vw(1),i(w(1),w(1))): ~b,i(vw(1),i(w(1),w(1))): ~a,i(vw(1), i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ @ (a+b),i(w(1),w(1)): ~ (@
branch 2
[i(w(1),w(1)): ~ @a,i(w(1),w(1)): @a+ $b]<---> [i(vw(1),i(w(1),w(1))): ~b,i(vw(1),i(w(1),w(1))): ~a, i(vw(1),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ @ (a+b),i(w(1),w(1
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(vw(1),i(w(1),w(1))): ~b, i(vw(1),i(w(1),w(1))): ~a,i(vw(1),i(w(1),w(1))): ~ (a+b),i(w(1),w
~ (@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(vw(1),i(w(1),w(1))): ~b, i(vw(1),i(w(1),w(1))): ~a, i(vw(1),i(w(1),w(1))): ~ (a+b), i(w(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(vw(1),i(w(1),w(1))): ~b, i(vw(1),i(w(1),w(1))): ~a, i(vw(1),i(w(1),w(1))): ~ (a+b), i(w(
[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(vw(1),i(w(1),w(1))): ~b,i(vw(1),i(w(1),w(1))): ~a,i(vw(1),i(w(1
i(w(1),w(1)): ~ @ (a+b),i(w(1),w(1)): (@a+ $b-> @ (a+b))]
literal
i(vw(1),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(vw(1),i(w(1),w(1))): ~b,i(vw(1),i(w(1),w(1))): ~a, i(vw(1),i(w(1),w(1))): ~ (a+b),i(w(1),w(1)): ~ @ (a+b), i(w(1),w(1)): ~(@a+ $b-> @ (a+b))]***closed
unsatisfiable  in  d
    in   200  msecs.
```