Documentation - Handling Unexpected Exceptions in Legal Reasoning

Thomas Ecobichon,
Mathieu CARPENTIER,
Sylvie DOUTRE,
Jean-Guy MAILLY

September 8, 2025

1 Simplification Assumptions

We consider $\mathcal{V}$, a vocabulary of propositional variables.
Let:

$$w_1, ..., w_n \in \mathcal{V}$$

$$p_1, ..., p_k, c_1, ..., c_l, s_1, ..., s_m, a, b \in \mathcal{V} \cup \neg\mathcal{V}$$

With $l, m, n, k \in \mathbb{N}$

Background Knowledge $\mathcal{W}$:

$\mathcal{W}$ is a set of formulas representing the *background knowledge*. In $\mathcal{W}$, we consider formulas of the following types:

- $a \Leftrightarrow b$
- $a \Rightarrow b$
- $a \Leftarrow b$
- $\neg(w_1 \wedge w_2) \wedge ... \wedge \neg(w_{p-1} \wedge w_p)$

Rule Base $\mathcal{R}$:

$\mathcal{R}$ is a rule base, where rules consist of premises and conclusions.
In $\mathcal{R}$, we consider rules of the form:

$$(p_1 \wedge ... \wedge p_k) \rightsquigarrow (c_1 \wedge ... \wedge c_l)$$

We assume that the premises of rules are consistent.

Hypothesis. $\forall r1, r2 \in \mathcal{R} \times \mathcal{R}, \quad r1 \Leftrightarrow r2$ *iff* $r1 = r2$.

$\mathcal{S}$:

$\mathcal{S}$ represents a situation of the form:

$$s_1 \wedge ... \wedge s_m$$

The premises of the situation are consistent.

2 Function and Class Documentation

2.1 Variable Management

2.1.1 Class `VariableUnicity`

This is a dictionary of propositional variables, implementing $\mathcal{V}$.
Methods:

- `.get(name)` : Returns the variable corresponding to the given name and adds it to the dictionary if it does not exist yet.
- `.get_many(*names)` : Generalized method for a list of variable names.

2.1.2 Function `list_to_vars(Var_dict,Str_List)`

Given a list of variable names (potentially with multiple negations), returns a list of corresponding variables.
Unlike `VariableUnicity`, this handles negations.

2.1.3 Function `ensemble_premices_equi2(premises, W)`

Augments a vector of premises with all variables implied by these premises according to $\mathcal{W}$ (synonyms and antonyms are taken care of before).

2.2 Rule Implementation

2.2.1 Class `Rule`

Defines a rule with premises and a conclusion.
Methods:

- `__str__` : Returns the rule as a string.

2.2.2 Function `is_a_in_b(short,long,W)`

Checks whether the premise list `short` is included in `long`.

2.2.3 Function `is_element_in_list(element,list, W)`

return the index of the element in the list if it is an element

2.2.4 Function `children_extraction(formula)`

Extracts all atomic variables from a formula. Used to determine if a formula from $\mathcal{W}$ is applicable in the current situation.

2.2.5 Function `update_dict_local(Rb,f)`

Updates the equivalence classes of synonyms/antonyms in `Rb.dict_local` based on a new equivalence formula `f`.
The method chosen to handle equivalence is that of ‘equivalence classes’. A dictionary is created that lists which variables are equivalent. Each time an equivalence is added to W, these classes are updated. Either by creating a new one if none of the parts of the equivalence already belong to a class. Or by joining two classes if the two parts are in different classes. Or by adding the missing part to the other class (same with the negation). Thus, in the system, when a premise that is part of an equivalence class is detected in a rule, it is replaced by the first one in the class, so that we can avoid processing all equivalences each time (for example, when comparing two rules, this avoids having to check each time whether the terms are synonyms. Since all synonyms have been translated with the same word, we know that the rules are equivalent if the premises and their conclusions are exactly the same).

2.2.6 Function `synonymes_elimination(premises,dict_local,W)`

Normalize a list of premises by eliminating synonyms and antonyms based on equivalence classes in `dict_local`

2.2.7 Function `dictionnaire_eval_rules(Rb,rules)`

Given a `Rule_base` and a vector of `Rule` (1 or 2 rules), returns a dictionary of all variables in the `Rule_Base`, the variables in the initialized situation $\mathcal{S}$, and sets the conclusions of the 2 rules to *True* (except negations). Other variables are set to *False*. Also returns the names of all propositions used. If premises are contradictory (a variable and its negation), returns -1 to indicate incompatibility with the current situation.

Remark. *This dictionary is later used to evaluate whether two rules are incompatible, i.e., if there exists a formula in $\mathcal{W}$ that is false given our set of premises.*

Remark. *This function is only used when there is an inclusion relation between two rules. Since rules are assumed consistent, we should not encounter conflicts between premise values.*

2.3 Class `Rule_base` Definition

Attributes:

- `premisses` : List of all used premises
- `conclusions` : List of all rule conclusions
- `rules` : List of rules in the base

- `P` : Rule matrix; each rule is represented by a vector of length equal to the total number of premises in the base. Presence = 1, negation = -1, absence = 0
- `C` : Vector of conclusions for each rule
- `compteur` : Number of rules
- `Var_dictionnary` : Variable dictionary to avoid duplicates
- `W` : Background knowledge
- `S` : Current situation
- `S_original` : Initial version of S (string)
- `rules_original` : Initial version of the rules (string) before adding implications, etc...
- `dict_local` : IMPORTANT : list which contains all the information about synonyms and antonyms (equivalence classes)

Methods:

- `__str__` : Returns all rules as a string
- `all_dictionnary` : Returns the dictionary for the rule base
- `add_W(f_string_list)` : Initializes $\mathcal{W}$ from a list of formulas (strings)
- `init_S(list_S)` : Initializes $\mathcal{S}$ from a list of premises
- `add_rules(list_P,list_C)` : Initializes and adds rules from lists of premises and conclusions
- `inclusion(indices)` : Returns indices of rules whose premises are included in the current situation (applicable rules). If called with $indices =$ [], compares with all rules.
- `compatibility_matrix(indices)` : Returns a compatibility matrix for the given rules. Entry $(i, j) = 1$ if rule j is an exception to rule i, 0 otherwise.
- `dist_hamming(indice)` : Returns a vector of Hamming distances between the given rule and all rules (including itself). Uses *scipy.spatial.distance.hamming.*
- `compatible(rule)` : Checks whether a rule is compatible with the situation $\mathcal{S}$, considering only relevant formulas in $\mathcal{W}$.

2.4 Creating Formulas from Strings

2.4.1 Function `get_var_from_index(token_to_var, i)`

Returns the variable corresponding to the index `i`.

2.4.2 Function `formula_creator(tokens, token_to_var)`

Creates a formula from tokens and variables, handling parentheses, negation, conjunction, and disjunction.

2.4.3 Function `extract_subtokens(tokens, start)`

Handles nested parentheses.

2.4.4 Function `eval_subformula(tokens, token_to_var,i,neg)`

Evaluates a subformula at index `i`.

2.4.5 Function `str_to_formula(formula_str,Rb)`

Transforms a string into a formula using the helper functions.

2.5 Rule Selection Using Hamming Distances

2.5.1 Function `Select_Rule(rulebase, regles_possibles)`

Returns rules corresponding to given indices.

2.5.2 Function `select_fct_threshold(Dist_vector, threshold)`

Returns indices of rules below a threshold (excluding the rule itself).

2.5.3 Function `select_fct_minimal(Dist_vector, threshold)`

Returns rules with minimal Hamming distance.

2.5.4 Function `select_fct_threshold_minimal(Dist_vector, threshold)`

Combines the previous two: selects the rule with minimal distance below a threshold.

2.6 Integration Functions

2.6.1 Function `scenario_check(S, rulebase, deja_appliquees)`

Finds applicable rules in situation $\mathcal{S}$, accounting for priorities and already applied rules. Eliminates rules whose conclusions conflict with $\mathcal{S}$. Returns the list, a log, and indices of rules in the base.

2.6.2 Function `choix_exception(distance_method, rulebase, selection_fct_and_args,regle_choisie)`

Selects rules from which to propose exceptions for `regle_choisie`, based on distance and selection method. Returns selected rules and associated exceptions.

2.6.3 Function `difference_premisses(longue, courte, W)`

Difference between 2 premises list (short list is a sublist of long list).

2.6.4 Function `exception(Rb, selected_indices)`

Filters rules without exceptions and returns the remaining rules with their exceptions.

2.7 Useful Functions

2.7.1 Function `call_llm(prompt, MODELS, clients, session)`

Calls the llm selected with the prompt.

2.7.2 Function `get_files(session)`

returns the path depending on the language selected (more explanation in comment in the code)

2.7.3 Function `get_distance_method()`

Just returns the different distance methods (only Hamming as of now). This way, the text is not loaded immediately and the translation with Babel works correctly.

2.7.4 Function `get_selection_method()`

Same with the selection method

2.7.5 Function `get_log(key)`

Same for some text in the log. The `key` just corresponds to which phrase is needed

2.7.6 Function `init_Rb(session)`

Initilizes the Rule Base with the rules contained on the selected Rule Base or in the default one if no Rule Base has been selected.

2.7.7 Function `get_prompt(scenario,premises,session)`

Returns the prompt in the langugage selected in session

2.7.8 Function `get_complement(S_join,complement,session)`

Complementary prompt for the llm.

2.7.9 Function `reset_session(session,keep_keys=None,reset_rules_updates=False))`

Resets the session except for the keys in keep keys.