A reasoning system for Defeasible Deontic Logic: HOUDINI

University of Verona

May 31, 2024

1 Introduction

The need for an online architecture able to reason and draw conclusions using Defeasible Logic has driven, in the past, the implementation of several tools, such as *SPINdle* [LG09] and others. However, for a variety of reasons, there is no current modern architecture, available as open source, and specifically available online, to treat this kind of knowledge. We therefore developed the architecture specified in this paper.

One of the most recent among such tools is *Houdini* [CGO+22], an online reasoning system for Defeasible Logic. In this paper, we discuss the implementation of its current version (2.0) that extends the previous one by including: (i) propositional logic, (ii) deontic logic, (iii) reparation chains, and (iv) numeric variables management (all features omitted, or only partially implemented, by the other technologies). The solution is presented from a functional and design viewpoint, while we briefly discuss the algorithmic solution underlying the technology itself.

The rest of the work is organised as follow. Section 2 introduces underlying logical framework, Section 3 discusses the architecture of the technology, while Section 4 provides an analysis of the data structures employed and the optimizations used in the solution.

2 Defeasible Deontic Logic

Defeasible Logic (DL) [Nut03] is a simple, flexible, and efficient rule-based non-monotonic formalism. Its strength lies in two main features. First, its constructive proof theory allows it to draw meaningful conclusions from (potentially) conflicting and incomplete knowledge base. In non-monotonic systems, more accurate conclusions can be obtained when more pieces of information become available. Second, many variants of DL have been proposed for the logical modelling of different application areas, specifically agents [KB15, GOS+16], legal reasoning [COR17], and workflows from a business process compliance perspective [OGSC13, COT16].

We start with its language. Let PROP be a set of propositional atoms, and Lab be a set of arbitrary labels (the names of the rules). Accordingly, the set PLit = PROP $\cup \{\neg p \mid p \in \text{PROP}\}$ is the set of *plain literals*. ModLit = $\{ml, \neg ml \mid l \in \text{PLit} \wedge m \in \{\mathsf{C}, \mathsf{O}, \mathsf{P}\}\}$ is the set of *modal literals*, while the set of *deontic literals* is restricted to $\{\mathsf{O}, \mathsf{P}\}$. Finally, the set of *literals* is Lit = PLit $\cup$ ModLit. The *complement* of a literal l is denoted by $\sim l$: if l is a positive literal p then $\sim l$ is $\neg p$, and if l is a negative literal $\neg p$ then $\sim l$ is p. We shall not have specific rules nor modality for prohibitions, as we will treat them according to the standard duality that something is forbidden iff the opposite is obligatory (i.e., $\mathsf{O}\neg p$).

Modality C is to derive *constitutive statements* (via *count-as rules*, see hereafter). Deontic literals represent prescriptive behaviours: $\mathsf{O}l$ says that 'l is obligatory', $\mathsf{P}l$ that 'l is permitted', $\sim\mathsf{O}l$ that 'l is not obligatory', and symmetrically $\sim\mathsf{P}l$ that 'l is not allowed. We shall not have a specific modality for *prohibitions* F, as we abide to the standard notion that "something (l) is forbidden iff the opposite of that something ($\sim l$) is mandatory", i.e. $\mathsf{F}l \equiv \mathsf{O}\sim l$. At the same level, modality P is to represent *strong permissions*, as proposed in [GORS13][1].

Definition 1 (Defeasible Deontic Theory) *A* defeasible deontic theory D *is a tuple* $(\mathrm{F}, R, >)$, *where* F *is the set facts,* R *is the set of rules, and* $>$ *is the* superiority relation.

[1] *Weak permissions* represent that "something (l) is permitted if the opposite ($\sim l$) is *not* mandatory', i.e. $\mathsf{P}l \equiv \sim\mathsf{O}\sim l$.

A theory is *finite* if the set of facts and rules are so. The set of facts $\mathrm{F} \in \mathrm{Lit}$ represent indisputable statements, piece of information that are to be considered always true and accepted without any inference, like "Chokchok is a wallaby" formally $wallaby(Chokchok)$. A defeasible deontic theory is meant to represent a normative system, where the rules encode the norms of the systems, and the set of facts corresponds to a case.

Rules R are three *types*: *strict rules*, *defeasible rules*, and *defeaters*. Rules are also of two *kinds*: *constitutive rules* (count-as rules) R^{C} model constitutive statements, whilst *deontic rules* model prescriptive behaviours: *obligation rules* R^{O} determine which obligations are in force, *permission rules* R^{P} represent *strong* (or *explicit*) permissions.

Lastly, $> \subseteq R \times R$ is the *superiority* (or *preference*) *relation*, which is used to solve conflicts in case of potentially conflicting information; $>$ is a binary and irreflexive relation.

Definition 2 (Rule) *A* rule *is an expression of the form* $r\colon A(r) \hookrightarrow_m C(r)$, *where*

1. $r \in \mathrm{Lab}$ *is the unique* name *of the rule;*
2. $A(r) \subseteq \mathrm{Lit}$ *is the set of antecedents;*
3. *An arrow* $\hookrightarrow \in \{\rightarrow, \Rightarrow, \rightsquigarrow\}$ *denoting, respectively, strict rules, defeasible rules, and defeaters;*
4. *The modality of the rule* $m \in \{\mathsf{C}, \mathsf{O}, \mathsf{P}\}$*;*
5. *The consequent* $C(r)$, *which is either (A) a single, plain literal* $l \in \mathrm{PLit}$, *if (i)* $\hookrightarrow \in \rightarrow, \rightsquigarrow$ *or (ii)* $m \in \{\mathsf{C}, \mathsf{P}\}$, *or (B) an* $\otimes$*-expression, if* $m \equiv \mathsf{O}$.

If $m = \mathsf{C}$ then the rule is used to derive non-deontic literals (constitutive statements), whilst if m is O or P then the rule is used to derive deontic conclusions (prescriptive statements). The conclusion $C(r)$ is a single literal is (i) the rule is strict or a defeater, or (ii) $m = \{\mathsf{C}, \mathsf{P}\}$; in case $m = \mathsf{O}$, then the conclusion is an $\otimes$-expression.

A *strict*, *constitutive*, *rule* is a rule in the classical sense: whenever the premises are indisputable, so is the conclusion. The statement "All wallabies are mammals" is hence formulated through the strict, constitutive rule $r1\colon wallaby(X) \rightarrow_{\mathsf{C}} mammal(X)$', as there is no exception to it. On the other hand, defeasible rules and defeaters represent the non-monotonic part of the logic. Specifically, defeasible rules are to conclude statements that can be defeated by contrary evidence; in contrast, defeaters are special rules whose only purpose is to prevent the derivation of the opposite conclusion. Accordingly, a prescriptive behaviour like "At traffic lights it is forbidden to perform a U-turn unless there is a 'U-turn Permitted' sign" can be formalised via the general obligation defeater '$r2\colon AtTrafficLight \rightsquigarrow_{\mathsf{O}} \neg UTurn$', and the exception via the defeasible permissive rule '$r3\colon UTurnSign \Rightarrow_{\mathsf{P}} UTurn$'.

Obligation rules gain more expressiveness with the *compensation operator* $\otimes$ for obligation rules, which is to model reparative chains of obligations. Intuitively, $a \otimes b$ means that a is the primary obligation, but if for some reason we fail to obtain, to comply with, a (by either not being able to prove a, or by proving $\sim a$) then b becomes the new obligation in force. This operator is used to build chains of preferences, called $\otimes$-expressions.

The formation rules for $\otimes$-expressions are: (i) every plain literal is an $\otimes$-expression, (ii) if A is an $\otimes$-expression and b is a plain literal then $A \otimes b$ is an $\otimes$-expression [GORS13].

In general an $\otimes$-expression has the form '$c_1 \otimes c_2 \otimes \cdots \otimes c_m$', and it appears as consequent of a rule '$A(\alpha) \hookrightarrow_{\mathsf{O}} C(\alpha)$' where $C(\alpha) = c_1 \otimes c_2 \otimes \cdots \otimes c_m$; the meaning opf the $\otimes$-expression is: if the rule is allowed to draw its conclusion, then c_1 is the obligation in force, and only when c_1 is violated then c_2 becomes the new in force obligation, and so on for the rest of the elements in the chain. In this setting, c_m stands for the last chance to comply with the prescriptive behaviour enforced by α, and in case c_m is violated as well, then we will result in a non-compliant situation.

The previous prohibition to perform a U-turn can also be represented via a defeasible rule that foresees a compensatory fine, like '$r2\colon AtTrafficLight \Rightarrow_{\mathsf{O}} \neg UTurn \otimes PayFine$', to be paid in case of an illegal U-turn.

A deontic defeasible theory contains all the elements the DL inference model needs in order to draw its *conclusions*. A *conclusion* in D is a *tagged literal*, with the following meaning:

- $+\Delta_m l$ means that l is *strictly* (or *definitely*) *provable* in D with modality m, because ml is a fact, or because there exists *strict proof* for it;

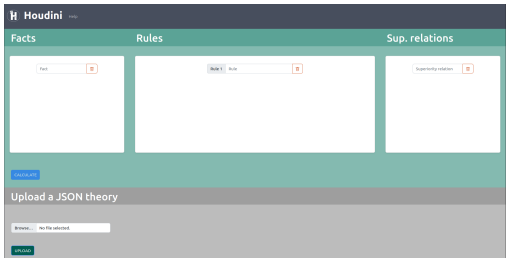

Figure 1: Houdini's home page

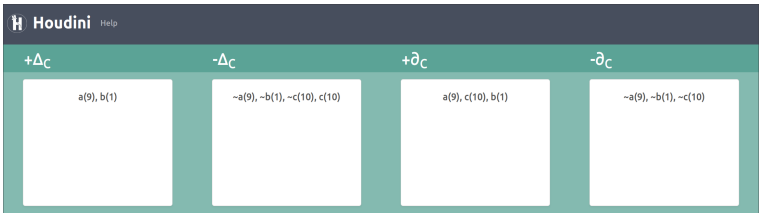

Figure 2: Houdini's computed extensions (detail)

- $-\Delta_m l$ means that l is *strictly refuted* in D with modality m, as l is *neither* a fact, *nor* there exists a strict proof;
- $+\partial_m l$ means that l is *defeasibly provable* in D with modality m, because either it was strictly proved[2];
- $-\partial_m l$ means that l is not defeasibly provable in D with modality m.

Note that $-\partial_m l$ is different from $+\partial_m \sim l$, as in the former case there does not exist a defeasible proof for ml, whereas in the latter case there exists a defeasible proof for $m\sim l$. For each of these tagging types, technical descriptions about the inference process, named *proof tags*, and relative demonstrations, are available and well described in [ABGM01].

3 Architecture of the technology

3.1 Technical details

At the current state, Houdini is a Java-based web-app with a single endpoint and a responsive and user-friendly UI. The choice of programming language immediately fell on Java due to its high maintainability and portability as well as the existence of mature web frameworks suitable to develop straightforward applications, that is, easy to understand for non-professional people such as students and researchers with a limited knowledge of programming.

In particular, Houdini is open-source, and available on request, and developed by using Java 11 and Spring Boot 2.7.1. The UI is simple Javascript, light and portable.

A single-page application design was considered but was later discarded taking into consideration the added memory footprint and complexity for a small gain, given the limited interaction required from the user. Houdini uses a custom parser (to read and validate the inputted theory) generated using ANTLR 4 parse tree listeners. ANTLR is a widely used and powerful parser generation framework that has great support, especially in Java: although it can be unwieldy and sometimes an overkill for projects that do not require most of its tools (such as ours), this advantage alone led us to prefer it to a possible custom-build parser in plain Java, which would be a much harder tool to maintain.

3.2 Data and UI

Houdini's UI comprises of two pages: the main, and the result, page (Figures 1 and 2). The main page allows to insert the defeasible deontic theory's specifics in two ways: (i) by hand, or (ii) by uploading

[2]If something is 'always' the case, then it is also 'typically' the case.

a JSON file.

Front-end checks with color-coded error messages guide users and let them correct ill-formed inputs, forbidden characters and plain wrong syntax. These controls are responsive and can fire an error message at the top of the screen with each single inserted character and mouse click. Input fields are generated dynamically and can be eliminated with the press of a button so as to have a clean interface devoid of too many unused spaces. Users are presented, at the beginning, with three different kinds of empty fields wherein they can insert the elements of the theory one-by-one, or an upload button which can be used to load a JSON file with the whole theory specified. The syntax required for this file is easy to understand and mimics the structure of a hand-inserted theory (Fig. 3).

Facts accept two kinds of elements: single literals (any alphanumerical string plus superscripts, underscores, and '∼', which can only be used to interpret the literal as a negation) or propositions composed by a variable name and, inside parentheses, their arguments (e.g. "∼a", "∼a(3,1,4,1,5)").

Rules are specific strings divided into a body (optional), an arrow, and a head. The body of a rule can be made of one or more elements chosen between literals, propositions (a variable name plus its parameters, e.g. "a(x)") and logical constraints[3] (such as "x < 10"), separated by commas. Moreover, literals and propositions can be prefixed by a tag indicating its deontic mode: [C] (constitutive), [O] (obligation) and [P] (permission). If no tag is present then the inferred mode is constitutive. The arrow is mandatory and defines the nature of the rule; it can also be marked by a deontic tag indicating the mode of the rule's head. There are three possibilities here: strict (−>), defeasible (=>) or defeater (∼>). The head is also mandatory. It can be a single literal or a proposition (a variable name plus parameter expressions). If the head's mode is O, marked in the arrow, then a reparation chain is possible and this is indicated by any combination of literals and propositions separated by the symbol "⊗".

As an example, the following rule is syntactically correct and is allowed by the system: "a, [O]b(x), x > 3 =>[P] ∼c(x+4)". The body is "a, [O]b(x), x > 3" and so it is made up of three elements: the literal "a", a proposition marked with the deontic tag O with variable name "b" and parameter "x" and a logical constraint "x > 3". The arrow is a defeasible arrow marked with the deontic tag P. The head is a proposition with variable name "∼c" (a negation) and parameter expression "x+4".

Superiority relations are simply two rules names separated by a ">" character: the left-side indicates the superior rule, whereas the right-side the inferior one. A rule name is simply the character "r" plus a number. For example, "r5 > r1" is a valid superiority relation. Once the user inserts the data Houdini parses and validates it, Houdini computes the extensions $\pm\#_m$, $\# \in \{\Delta, \partial\}$, $m \in \{\mathsf{C}, \mathsf{O}, \mathsf{P}\}$, and displays them in an appropriate result page.

3.3 Parsing

Reasoning about and computing the extension sets of a DDL theory requires, first, a clear and precise representation of it. Hence, Houdini expects users to provide data which complies with a precise syntax that cannot lead to any ambiguity. In particular a theory must be provided by following a context-free grammar specification which is then used by Houdini's custom parser (generated using ANTLR 4 parse tree listeners) to validate it, server-side. This is the first operation done on the inputted data (see Fig. 4).

If the input is ill-formed, an error is returned. Client-side checks are also performed to avoid unnecessary parsing, and to deliver quicker feedback. However, due to the grammar specification being context-free (hence harder to validate than simple regular expression strings) and the design decision to have a light and responsive front-end, the client-side checks do not encompass all the necessary steps to formally verify the adherence of the input to the specification.

Houdini's parser not only validates the user data but also builds, incrementally, as it traverses the tree, the necessary internal data structures that are later used by the reasoner to compute the extensions. This is done to be more computationally efficient. Some optimization tricks are also employed, both on the grammar side, such as grouping and prioritizing the most common expressions (chosen empirically), and on the parser side, such as parsing superiority relations before everything else and keeping note of which rules they refer to so as to, later in the rules parsing phase, avoid keeping in memory too many rules but only those that appear in some superiority relation (which are usually way fewer than the total number of rules). Moreover, mathematical expressions (which can appear inside a

[3]Logical constraints must be evaluated as true for the rule to be applicable.

⟨*word*⟩ ::= Any alphanumeric string possibly with superscripts and underscores
⟨*integer*⟩ ::= Integer numbers
⟨*fact*⟩ ::= [‘∼’], ⟨*word*⟩, [‘`(`’, ⟨*args*⟩, ‘`)`’]
⟨*args*⟩ ::= ⟨*integer*⟩ | ⟨*integer*⟩, ‘`,`’, ⟨*args*⟩
⟨*rule*⟩ :: = [⟨*body*⟩], (‘`->`’|‘`=>`’|‘∼`>`’), [⟨*modal*⟩], ⟨*head*⟩
⟨*body*⟩ ::= ⟨*bElem*⟩ | ⟨*bElem*⟩, ‘`,`’, ⟨*body*⟩
⟨*bElem*⟩ ::= ⟨*bodyLiteral*⟩ | ⟨*logicExpression*⟩
⟨*bodyLiteral*⟩ ::= [‘∼’], [⟨*modal*⟩], [‘∼’], ⟨*word*⟩, [‘`(`’, ⟨*params*⟩, ‘`)`’]
⟨*modal*⟩ ::= ‘`[C]`’| ⟨*deontic*⟩
⟨*deontic*⟩ ::= ‘`[O]`’|‘`[P]`’
⟨*params*⟩ ::= ⟨*word*⟩ | ⟨*word*⟩, ‘`,`’, ⟨*params*⟩
⟨*logicExpression*⟩ ::= ⟨*exp*⟩, ⟨*logicOp*⟩, ⟨*exp*⟩
⟨*logicOp*⟩ ::= ‘`>`’ | ‘≥’ | ‘`<`’ | ‘≤’ | ‘`!=`’ | ‘`==`’
⟨*exp*⟩ ::= ⟨*exp*⟩, ⟨*mathOp*⟩, ⟨*exp*⟩ | ‘`(`’, ⟨*exp*⟩, ‘`)`’ | ⟨*integer*⟩ | ⟨*word*⟩
⟨*mathOp*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`^`’
⟨*head*⟩ ::= ⟨*headLiteral*⟩ | ⟨*headLiteral*⟩, ‘∘’, ⟨*head*⟩
⟨*headLiteral*⟩ ::= [‘∼’], ⟨*word*⟩, [‘`(`’, ⟨*exps*⟩, ‘`)`’]
⟨*exps*⟩ ::= ⟨*exp*⟩ | ⟨*exp*⟩, ‘`,`’, ⟨*exps*⟩
⟨*supRel*⟩ ::= ‘`r`’, ⟨*integer*⟩, ‘`>`’, ‘`r`’, ⟨*integer*⟩

Figure 3: Partial description of the accepted syntax (EBNF)

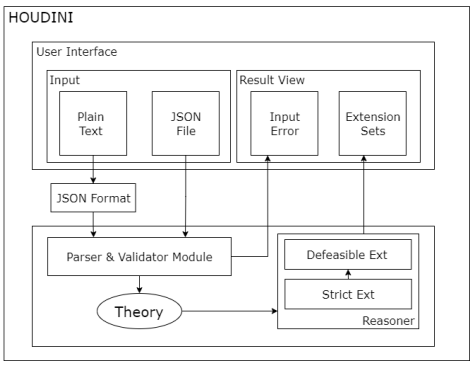

Figure 4: Houdini's architecture and data flow

rule head proposition) and logical expression (which can appear inside a rule body logical constraint) are transformed in their respective Reverse Polish Notation (RPN) versions to speed things up later when these expressions must be evaluated with actual arguments, multiple times; RPN expressions are easily evaluated and with minimal memory requirements.

3.4 Reasoner

The reasoner module implements the core functionality of Houdini; it is performed after the parsing phase: computing the complete extension of the given defeasible deontic theory. It comprises of two sub-modules: Strict Reasoner and Defeasible Reasoner.

The Strict Reasoner is responsible for computing both $\pm\Delta$ (for deontic modalities); it employs an injection and deactivation strategy to quickly find literals that belongs to these sets. *Injecting* a literal means that, when it gets put into $+\Delta$ or $+\partial$, all rules in which it appears in the body are marked (in principle, the literal is removed from the body, refer to Section 4.2 for specific details.) The same applies for the *deactivating* action but in this case the event is the literal being put into $-\Delta$ or $-\partial$. It starts from the facts, immediately added to $+\Delta$ and injected: if new injectable literals are found (heads of applicable strict rules, i.e. with empty rule bodies) the process continues until no new injectable literals are found. All literals, that which are neither facts, nor heads of strict rules,

are deactivated, and a similar process is carried out, using heads of discarded strict rules. At the end, $\pm\Delta$ are completely populated.

Defeasible Reasoner, instead, builds $\pm\partial$ (again, for all deontic modes). In this case the process is more involved as the membership conditions are more complex. Generally, it starts from a set of "candidates" for each ∂ extension set and an injection and deactivation set (whose elements which are immediately injected and deactivated). It loops over the candidates and checks if they satisfy the various membership conditions: when it finds a literal that does satisfy a condition it gets added to one of the two sets. However, injecting or deactivating a literal into rule bodies is not sufficient, generally, for adding their heads directly to the injection or deactivation sets (unlike in the strict phase): they are stored, usually, in a temporary set and later checked again. The process continues until the fixed point is reached: this happens the moment an entire Defeasible Resasoner loop goes without a change in either set: no literal has been injected or deactivated. The full $\pm\partial$ extensions are then returned and the algorithm ends.

4 Data structures and optimization

The main algorithm, the reasoner module, acts on Literal and Rule instances which have been initialized by the parser and which hold, together with a general Theory container instance, all the necessary information to work with. For example, Literals hold, among many other information, references to their opposites, all those rules wherein they appear as heads and as elements of their bodies, and whether they have been, at any moment, been assigned by the reasoner to an extension set ($\pm\Delta$ and $\pm\partial$) or are still undecided, and their deontic mode.

Rules, instead, keep track of all those literals which appear in their bodies and heads, the deontic mode of the head, parameters, logical expressions, and the order in which they appear in their heads' mathematical expressions, etc. Also, all these classes implement methods called by the reasoner to compute the extensions.

The decision of having a strict pairing between data and methods that can act inside objects is due to the need of having to use similar but not identical functions depending on the type of the literal (normal or proposition), its mode, the presence of mathematical expressions in the rule's head etc.

4.1 Data structure: literals

Literals are the fundamental brick objects in Houdini architectural design. They hold more information than any other class and have specialized methods, used in the reasoner. An actual Literal instance represent a single literal such as "a" that may appear anywhere in a DDL theory. Literal has two subclasses: Variable, which represents a proposition such as "a(x)" or "b(x+y-3)" that may appear in rules, and AppliedVariable which represents "actualized"/"applied"/"grounded" variables such as "a(2)" or "a(2718)" that may only appear among facts and which are injected, by the reasoner, in propositions with the correct arity that may appear in rules' bodies (so these two AppliedVariables will be injected in any rule with "a(x)" in their body). The main fields of a Literal class are:

- *label*, *type*, *mode*: the name, the type (not a negation or a negation) and the deontic mode (C, O, or P). So for example if these three fields are "a", "NotNegation" and "P" this will represent the literal "[P] a"; these are needed to uniquely identify a literal;
- *opposite*, *CLiteral*, *OLiteral* and *PLiteral*: references to their opposite literal (if not a negation, its negation and vice versa) and to the same literal but with, respectively, a C, a O, and a P mode[4];
- *headOf*: a list of references to all rules wherein this literal appears in their head;
- *bodyOf*: a list of references all rules wherein this literal appears in their body;
- *DeltaState* and *PartialState*: undecided, Plus or Minus. This represents in which set (if any) they have been assigned by the reasoner. Depending on this literal's mode, these fields will represent the literal belonging to the respective $\pm\Delta_\diamond$ or $\pm\partial_\diamond$ set where $\diamond = \{C, O, P\}$;

[4]Houdini considers separately two literals with the same name, but different modes.

- three integers variables indicating the number of rules (differentiating based on type: strict, defeasible and defeater) the literal is head of. These are used by the reasoner to keep track of discarded rules;
- two Booleans indicating (based on rule type: strict + defeasible, and defeater). These are used to mark a literal as definitely activated (i.e. in $+\Delta$ or $+\partial$).

Most of these fields are for efficiency reasons: a larger memory footprint and a bit of data redundancy (the references between different literals and between literals and rules are cross-references) as a trade-off for computational gains in the reasoner. For example, having inside a Literal instance the two lists *headOf* and *bodyOf* make it easy to mark rules as applicable or discarded: if a literal is put into $-\Delta$ or $-\partial$, then activating (mark them as applicable) or deactivating rules (mark them as discarded) simply requires looping over *bodyOf*; checking a literal activation can be done by checking the status of all rules referenced in headOf. As for the Literal class methods, we mention:

- *checkPlusPartialCondition* and *checkMinusPartialCondition*: helper methods used by the reasoner to check $\pm\partial$ membership conditions. These are different based on the literal mode[5];
- *injectIntoRules* and *deactivateRules*: called by the reasoner when the literal being put into an extension set requires to activate or deactivate (mark as applicable or discarded) all rules wherein it appears in the body.

The Variable class contains all the information of its superclass Literal plus an integer *arity* representing the number of parameters. So, "a(x)" is actually different than "a(x,y)": of *arity* 1 and 2 respectively (but "a(x)" and "a(y)", and likewise for any other parameter name, are considered the same. See next paragraph.) Importantly, it contains a method called *apply* which instantiates, based on a set of arguments whose amount must be equal to its arity, an AppliedVariable instance with those arguments as fields. The AppliedVariable class is like a Variable (its actual superclass) but with concrete arguments and a reference to the original variable (a single Variable instance from which it derives). So "a(1,2)" can derive from "a(x,y)" and it keeps a reference to it: this is used to link applied variables to all those rules in which they may be injected or discarded. It also overrides the *injectIntoRules* method from Literal: the injection of an AppliedVariable instance inside a rule is trickier as it requires to "pass" and memorize its arguments to compute then, eventually, the rule head mathematical expression which may depends on that instance concrete arguments.

4.2 Data structure: rules

A Rule instance represents a single logical rule of a theory such as "a(x), b(y) $\Rightarrow$ c(y+x)" or "a $\rightarrow$[O] z". The mode of a rule's head and its type (strict, defeasible or defeater) are stored in two fields along with an integer representing the rule's id (which is used to uniquely identify it). References to the head literal (or proposition) and all its body elements are also present for easy access during the reasoning phase. An integer field, *remaining*, indicates the number of body literals (or propositions) that have yet to be activated: this is used to determine whether the rule is applicable or not (when the counter drops to zero a rule is applicable.) Superiority relations do not require another class: they are internally represented by cross-references stored inside rules (*winsAgainst* and *losesAgainst*).

4.2.1 Mathematical and logical expressions handling

The way parameters, mathematical and logical expressions are managed is quite involved. Strictly speaking, a rule body can only contain propositions (i.e. variables such as "a(x,y)") or logical expressions (such as "x > 3"), whereas a rule head, instead, can contain propositions with or without mathematical expressions (such as "g(x-y)" or "g(z)").

Expressions are simply a list of strings, i.e. a string for each operator and variable name, following the reverse polish notation (transformed in the parser): so, for example, the list ["x", "y", "+"] corresponds to "x + y". All rules store their logical expressions and mathematical expressions in two fields, *gates* and *headExpressions* respectively, which are used when they need to be evaluated.

[5]There are not equivalent for $\pm\Delta$: their membership conditions are more easily checked by the reasoner without resorting to helper functions.

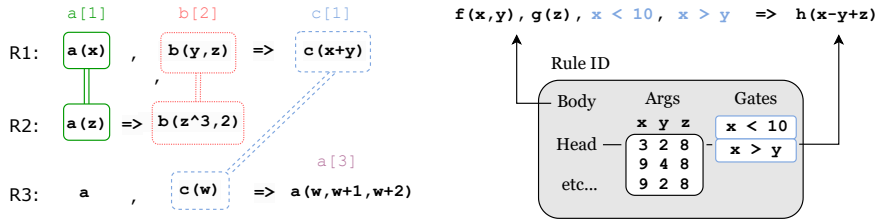

Figure 5: On the left, the internal linking in a three-rules theory with variables; on the right, a schematic example of the state of a rule with two gates (top) after the injection of three elements (f(3,2), g(8), f(9,4)).

The system considers propositions with identical name and arity the same, regardless of its parameters names: internally, they are represented as "name[arity]" and during the parsing phase they are all linked together (Fig. 5, left panel). It follows that parameters names and the specific order in which they appear in the body, as well as the name of the propositions in which they appear, must be stored individually for all rules. So, for example, rule number 1 in the figure keeps the information that "a[1]" will pass "x", whereas "b[2]" will pass "y" and "z", in that specific order. This is necessary to correctly compute the head expressions with concrete arguments.

The linking between variables is necessary for the system to know where to inject "applied variables" (which hold a reference to the original proposition/variable) after a rule firing: this cannot be known at parsing time for all possible "actualization" of a variable, because they depend on the initial facts and on how rules interact with each other, so the system has to track over time, for each rule, what arguments have already been passed. Let us consider, again, Fig. 5 as an example. Assume "a(2)" has been assigned to $+\partial$ and then injected in the bodies of rule 1 and rule 2; rule 2 fires so the variable "b[2]" with concrete arguments "8" and "2", i.e. the applied variable "b(8,2)", needs to be injected wherever "b[2]" appears in a rule's body. The applied variable holds information about the original (parent) variable so the linking is then used to inject it in rule 1 which can now fire with "c(10)", then injected in rule 3.

A set of arguments is stored for each rule. This is needed to track, over time, the concrete arguments passed by variables. Every time something is passed, a Cartesian product between the new arguments and the old ones is computed and, if gates are present, these new combinations are checked against the logical expressions and discarded if any evaluates to false. Take Fig. 5, right panel, as an example. The rule "f(x,y), g(z), x < 10, x > y => h(x-y+z)" presents two gates. Assume three elements, "f(3,2)", "g(8)" and "f(9,4)", are passed in this specific order. After the first element, the set of arguments ("Args") contains "{x: 3, y: 2}" and the rule cannot fire, even though the logical constraints are met. Then "g(8)" is injected: at this moment, Args contains "{x: 3, y: 2, z: 8}" and the rule can fire with "h(9)". When "f(9,4)" is injected, a Cartesian product between the new arguments and the old ones is computed, returning (the new elements are underlined): "{x: 9, y: 4, z: 8}", "{x: 9, y: 2, z: 8}" and "{x: 3, y: 4, z: 8}". The latter is discarded as the second logical constraint is not met; the rule fires with only two values: "h(13)" and "h(15)".

4.3 Optimizations

Most of the architectural choices, such as what kind of information to have in what classes, how to effectively represent rule activation/deactivation internally and parsing technicalities, were taken with computational complexity in mind; still, memory consumption has been optimized by limiting the amount of redundant references and data stored after the parsing phase, and by aggregating object instances that could be reduced to more general ones.

One common operation is activating/deactivating rules based on the status of their body literals. Hence, linking literals and all those rules wherein they appear in the body optimally is a must: this is done with a Hash Table (one for every literal) so as to have constant lookup times.

One of the first conundrums was how to keep track of the activation of a rule: because this depends on the status of *all* its body literals, activating a single one is, generally, not sufficient. We could theoretically check all others' statuses when a literal's status changes, but this would have an unnecessary cost of $O(n^2)$, where n is the number of literals in the body. Actually, the best way is to

just use a single counter inside a Rule instance, decremented each time a literal is activated: not only this does not require to keep track which literals have been activated already, but also the activation condition depends only on a simple integer check. The only problem is that we need to guarantee that every literal only activates a rule a single time: this is less trivial than one may expect and multiple checks and "safety measures" are performed. Still, a great amount of time is saved with this solution as checking literals' statuses is one the most performed operation by the algorithm. Deactivation is tracked similarly, with a counter that is decremented over time. Also, due to the fact that when a literal is activated or deactivated its status is set forever, we can simply memorize this information in a Boolean variable.

To avoid multiple passes on theory data, all necessary structures and object instances are built during the parsing phase, incrementally. This can be done without any backtracking and minimal temporary data storage. Superiority relations are parsed first: rule indexes are saved for later when the respective Rule instances need to be linked together, without the need of a Superiority Relation class. Then, rules are parsed: for every literal/proposition in the body and head an instance is created and is saved in a Hash Table (if later the same literal is referenced again in the theory, the existing instance is retrieved avoiding duplicates), its mode and type are saved and all the linking between the rule and literals is done. Expressions are transformed in their reverse polish notation. Facts are simply instantiated based on whether they are literals or proposition with concrete arguments.

References

[ABGM01] Grigoris Antoniou, David Billington, Guido Governatori, and Michael J. Maher. Representation results for defeasible logic. *ACM Trans. Comput. Log.*, 2(2):255–287, 2001.

[CGO+22] Matteo Cristani, Guido Governatori, Francesco Olivieri, Luca Pasetto, Francesco Tubini, Celeste Veronese, Alessandro Villa, and Edoardo Zorzi. Houdini (unchained): An effective reasoner for defeasible logic. In *AI³@AI∗IA*, volume 3354 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[COR17] Matteo Cristani, Francesco Olivieri, and Antonino Rotolo. Changes to temporary norms. In *ICAIL*, pages 39–48. ACM, 2017.

[COT16] Matteo Cristani, Francesco Olivieri, and Claudio Tomazzoli. Automatic synthesis of best practices for energy consumptions. In *IMIS*, pages 154–161. IEEE Computer Society, 2016.

[GORS13] Guido Governatori, Francesco Olivieri, Antonino Rotolo, and Simone Scannapieco. Computing strong and weak permissions in defeasible logic. *J. Philos. Log.*, 42(6):799–829, 2013.

[GOS+16] Guido Governatori, Francesco Olivieri, Simone Scannapieco, Antonino Rotolo, and Matteo Cristani. The rationale behind the concept of goal. *Theory Pract. Log. Program.*, 16(3):296–324, 2016.

[KB15] Kalliopi Kravari and Nick Bassiliades. A survey of agent platforms. *JASSS*, 18(1):11, 2015.

[LG09] H.-P. Lam and G. Governatori. The making of SPINdle. In G. Governatori, J. Hall, and A. Paschke, editors, *Rule Representation, Interchange and Reasoning on the Web*, number 5858 in LNCS, pages 315–322. Springer, 2009.

[Nut03] Donald Nute. Defeasible logic. In *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, volume 2543, page 151 – 169, 2003.

[OGSC13] Francesco Olivieri, Guido Governatori, Simone Scannapieco, and Matteo Cristani. Compliant business process design by declarative specifications. In *PRIMA*, volume 8291 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2013.