UNIVERSITY OF CAPE TOWN
DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2021

Title: Scalable Defeasible Reasoning

Author: Aidan Bailey

Project Abbreviation: SCADR

Supervisor(s): Tommie Meyer

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	10
Experiment Design and Execution	*0*	*20*	5
System Development and Implementation	*0*	*20*	15
Results, Findings and Conclusions	*10*	*20*	15
Aim Formulation and Background Work	*10*	*15*	15
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

Scalable Defeasible Reasoning

Aidan Bailey
University of Cape Town
Cape Town, South Africa
BLYAID001@myuct.ac.za

ABSTRACT

Knowledge Representation and Reasoning (KRR) is an approach to Artificial Intelligence (AI) in which a system is given information about the world (a Knowledge Base), with which it is able to draw conclusions through automated reasoning (entailment checking). The classical forms of reasoning are monotonic, in that by adding new information to a Knowledge Base, possible inferences cannot be retracted, only introduced. This is problematic, as when more information is added to a Knowledge Base, there is an increasing chance that a contradiction will be introduced, rendering classical forms of reasoning unreliable. Defeasible reasoning is a non-classical, non-monotonic form of reasoning, enabling systems to more reliably reason with Knowledge Bases containing contradictions, a process more akin to everyday human-based reasoning. Currently, no programming framework exists that explicitly supports defeasible entailment checking for Propositional Logic. We aim to implement and investigate the scalability of defeasible entailment algorithms. We also aim to implement a Knowledge Base generation tool to be used for testing the implementations.

CCS CONCEPTS

• **Theory of computation → Automated reasoning**; • **Computing methodologies → Nonmonotonic, default reasoning and belief revision**.

KEYWORDS

artificial intelligence, knowledge representation and reasoning, defeasible reasoning, satisfiability solving

1 INTRODUCTION

There are two main approaches to Artificial Intelligence (AI): Machine Learning (ML) and Knowledge Representation and Reasoning (KRR) [2]. Our research focuses on the latter. Knowledge Representation refers to the use of formal notation to express information about the world. Reasoning is the notion of drawing inferences from this information, with the key idea being to automate the process. Logics [1] is an umbrella term for the various Knowledge Representation and Reasoning subtypes, each utilizing differing syntax and semantics allowing for different kinds of logical relationships to be easily expressed. We will be focusing on one such type that represents propositional logical relationships (e.g., "the Earth is round"), known as Propositional Logic [1].

Propositional Logic [1] is monotonic, meaning that when new information is gathered, any previously possible inferences cannot be retracted, even if the new information introduces contradictions. This propends to the notion that Propositional Logic requires complete knowledge (propositions that are certainly true) in order to reason reliably. This is clearly not a "common-sense" approach to reasoning [3], as humans often (if not always) reason without complete knowledge. Thus, using Propositional Logic limits our ability to model human reasoning as the rational mind is not monotonic in nature.

For example, if a human has never encountered flightless birds, "all birds fly" would be a reasonable classical proposition. When they are told that "penguins are birds" and that "penguins do not fly", a contradiction is apparent, The monotonic mind would infer that "penguins do not exist", an undesired conclusion showcasing the unreliability of reasoning when under the assumption that one is working with complete knowledge. If the human instead made an initial defeasible proposition that "all birds *typically* fly", the non-monotonic mind would conclude that penguins represent an exception to this and thus, would retract any inferences derived from it when reasoning with penguins, ultimately leading to more reliable conclusions.

We make use of an extension of Propositional Logic where *typical* logical relations are allowed.

2 BACKGROUND

2.1 Propositional Logic

2.1.1 Syntax. The syntax of the Propositional Logic language $\mathcal{L}$ consists of propositional *atoms* and *Boolean connectives*. Atoms represent natural language propositions and are often denoted using abbreviations of their respective propositions (e.g., "Is a Dog" could be `isDog`, `dog`, `d`, ...) or using Greek characters (e.g., α,β,γ...). Boolean connectives represent logical relations among the atoms. There are five basic connectives in $\mathcal{L}$ as described by the table below.

Symbol	Name	Operand Type
$\neg$	Negation	Unary
$\wedge$	Conjunction	Binary
$\vee$	Disjunction	Binary
$\rightarrow$	Implication	Binary
$\leftrightarrow$	Equivalence	Binary

The combination of atoms and connective produce propositional *formulas*. Thus, the grammar of $\mathcal{L}$ can be defined recursively for a finite set of atoms $\mathcal{P}$ as follows:

- If α is an atom in $\mathcal{P}$, α is a *formula* of $\mathcal{L}$.
- If α and β are formulas of $\mathcal{L}$, then so are $(\neg\alpha)$, $(\alpha\wedge\beta)$, $(\alpha\vee\beta)$, $(\alpha \rightarrow \beta)$ and $(\alpha \leftrightarrow \beta)$.

2.1.2 Semantics. A propositional formula is evaluated using a propositional *interpretation*. An interpretation is a function $\mathcal{I} : \mathcal{P} \rightarrow \{\mathcal{T}, \mathcal{F}\}$, where $\mathcal{P}$ is a finite set of atoms, $\mathcal{T}$ is a truth-value of true and $\mathcal{F}$ is a truth-value of false. Once a interpretation has been applied, the formula is evaluated based on its Boolean connectives with the resounding truth-value being the result. If $I(\alpha)$ is true for some formula α, then $\mathcal{I}$ *satisfies* α, denoted $\mathcal{I} \Vdash \alpha$, and $\mathcal{I}$ is referred to as a *model* of α. If α has at least one model, α is said to be *satisfiable*. If α has no models, α is unsatisfiable and contains a contradiction. The set containing all models of α is denoted $Mod(\alpha)$. A propositional *knowledge base* is a finite set of formulas. For some knowledge base $\mathcal{K}$ and interpretation $\mathcal{I}$, $\mathcal{I}(\mathcal{K})$ is true if $\forall \alpha \in \mathcal{K}$, $\mathcal{I} \Vdash \alpha$.

If some formula α is true for every model of $\mathcal{K}$ (i.e., if $Mod(\mathcal{K}) \subseteq Mod(\alpha)$), then $\mathcal{K}$ *entails* α, written $\mathcal{K} \models \alpha$. Owing to the monotonic nature of propositional entailment, if $\mathcal{K}$ contains even a single contradiction, $Mod(\mathcal{K})$ will be empty and thus, $\mathcal{K}$ will entail anything.

2.2 KLM Approach to Defeasible Reasoning

2.2.1 KLM-framework. *Defeasible reasoning* is a form of reasoning which allows one to reason about knowledge bases which contain seemingly contradictory information, and allows for exceptions to general assertions. The KLM-approach [10, 12] provides a framework that introduces the *defeasible implication* in order to deal with statements such as "α typically implies β", written formally as "$\alpha \mid\!\sim \beta$", meaning if α is true, then this is typically enough information to believe β is also true. A detailed overview of this approach is provided by Kaliski in [9]. It is important to note that the defeasible implication represents a relationship between two propositional formulas and is not an extension to $\mathcal{L}$, i.e., formulas such as "$\alpha \mid\!\sim \beta \mid\!\sim \gamma$", "$\alpha \mid\!\sim (\beta \mid\!\sim \gamma)$" and "$\alpha \wedge (\beta \mid\!\sim \gamma)$" are not allowed.

2.2.2 Defeasible Entailment. The notion of defeasible entailment, denoted $\approx$, is not unique, i.e., there are many acceptable ways to infer information from a *defeasible knowledge base* (a knowledge base containing statements of the form $\alpha \mid\!\sim \beta$). The KLM approach suggests that any defeasible entailment method should adheres to several *rationality properties* proposed by Lehmann and Magnidor [12]. Giovanni et al. [8] proposed *LM-rational* as a term to describe any defeasible entailment procedure that satisfies all of these properties.

The rationality properties, defined by [9], for all knowledge bases $\mathcal{K}$ and propositional formulas α, β, γ as follows:

$$\text{(Ref) } \mathcal{K} \mid\!\approx \alpha \mid\!\sim \alpha \qquad \text{(And) } \frac{\mathcal{K} \mid\!\approx \alpha \mid\!\sim \beta, \mathcal{K} \mid\!\approx \alpha \mid\!\sim \gamma}{\mathcal{K} \mid\!\approx \alpha \mid\!\sim \beta \wedge \gamma}$$

$$\text{(LLE) } \frac{\mathcal{K} \mid\!\approx \alpha \leftrightarrow \beta, \mathcal{K} \mid\!\approx \alpha \mid\!\sim \gamma}{\mathcal{K} \mid\!\approx \beta \mid\!\sim \gamma} \qquad \text{(Or) } \frac{\mathcal{K} \mid\!\approx \alpha \mid\!\sim \gamma, \mathcal{K} \mid\!\approx \beta \mid\!\sim \gamma}{\mathcal{K} \mid\!\approx \alpha \vee \beta \mid\!\sim \gamma}$$

$$\text{(RW) } \frac{\mathcal{K} \mid\!\approx \alpha \rightarrow \beta, \mathcal{K} \mid\!\approx \gamma \mid\!\sim \alpha}{\mathcal{K} \mid\!\approx \gamma \mid\!\sim \beta} \qquad \text{(CM) } \frac{\mathcal{K} \mid\!\approx \alpha \mid\!\sim \gamma, \mathcal{K} \mid\!\approx \alpha \mid\!\sim \beta}{\mathcal{K} \mid\!\approx \alpha \wedge \beta \mid\!\sim \gamma}$$

Rational Closure [12] and Lexicographic Closure [11] are two approaches to defeasible reasoning which are LM-rational, with *Relevant Closure* [4] and *Ranked Entailment* [12] are two which are not [5]. This paper will focus on the first two.

2.3 Defeasible Entailment

Rational Closure is the most conservative form of defeasible entailment (i.e., it infers very little from a defeasible knowledge base), and can be defined both semantically and algorithmically, as laid out in [5].

2.3.1 Semantic Definition. A *ranked interpretation* is simply a ranking of all interpretations in $\mathcal{W}$ (the set of all possible interpretations), in order of typicality, starting at rank 0 (indicating the most typical interpretations) and ending at rank n, (the least typical, but are still plausible interpretations), followed by a single infinite rank (impossible interpretations). A ranked interpretation cannot contain empty ranks. Formally, a ranked interpretation is a function $\mathcal{R} : \mathcal{W} \rightarrow \mathbb{N} \cup \{\infty\}$, such that $\mathcal{R}(\mathcal{I}) = 0$ for some $\mathcal{I} \in \mathcal{W}$, and for every $r \in \mathbb{N}$, if $\mathcal{R}(\mathcal{I}) = r$, then for every j such that $0 \leq j \leq r$, there is an $\mathcal{I} \in \mathcal{W}$ for which $\mathcal{R}(\mathcal{I}) = j$.

$\mathcal{R}$ *satisfies* a formula α if α is true in all non-infinite ranks of a ranked interpretation $\mathcal{R}$, denoted $r \Vdash \alpha$. For defeasible statements (statements which use the $\mid\!\sim$ operator), we say $\mathcal{R}$ *satisfies* $\alpha \mid\!\sim \beta$ if in the lowest rank (the most typical rank) where α holds, β also holds.

We can construct an ordering on the set of all possible ranked interpretations for a knowledge base $\mathcal{K}$, where $\mathcal{R}_1 \leq_{\mathcal{K}} \mathcal{R}_2$ if for every interpretation $\mathcal{I} \in \mathcal{W}, \mathcal{R}_1(\mathcal{I}) \leq \mathcal{R}_2(\mathcal{I})$. There is a unique minimal element $\mathcal{R}_m$ of the ordering $\leq_{\mathcal{K}}$ such that for every other ranked interpretation $\mathcal{R}_i$ in the set of all ranked interpretations for a given knowledge base, $\mathcal{R}_m \leq_{\mathcal{K}} \mathcal{R}_i$, as shown in [7]. If $\mathcal{R}_m \Vdash \alpha \mid\!\sim \beta$, then we say $\mathcal{K}$ defeasibly entails $\alpha \mid\!\sim \beta$ (denoted $\mathcal{K} \approx \alpha \mid\!\sim \beta$).

2.3.2 Base Rank Algorithm. Before showing the algorithm, we define the *materialisation* of a knowledge base $\mathcal{K}$ as

$$\overrightarrow{\mathcal{K}} = \{\alpha \rightarrow \beta : \alpha \mid\!\sim \beta \in \mathcal{K}\}$$

and $\mathcal{E}$ as a finite set of formulas.

```
(1) $i = 0$
(2) $\mathcal{E}_i = \overrightarrow{\mathcal{K}}$
(3) $\mathcal{E}_{i+1} = \{\alpha \rightarrow \beta \in \mathcal{E}_i | \mathcal{E}_i \models \neg\alpha\}$
(4) if $\mathcal{E}_{i+1} \not\equiv \mathcal{E}_i$:
    (a) $\mathcal{R}_i = \mathcal{E}_i \setminus \mathcal{E}_{i+1}$
    (b) $i = i + 1$
    (c) go to (3)
(5) $\mathcal{R}_\infty = \mathcal{E}_i$
(6) $n = i$
(7) Return $(\mathcal{R}_0, \ldots \mathcal{R}_{n-1}, \mathcal{R}_\infty, n)$
```

2.3.3 Defeasible Entailment Checking. Given a knowledge base $\mathcal{K}$, the result from Base Rank $(\mathcal{R}_0, \ldots \mathcal{R}_{n-1}, \mathcal{R}_\infty, n)$, and some defeasible implication query $\alpha \mid\!\sim \beta$.

```
(1) $i = 0$
(2) $\mathcal{R} = \bigcup_{i=0}^{j<n} \mathcal{R}_j$
(3) if $\mathcal{R} \cup \mathcal{R}_\infty \models \neg\alpha$:
    (a) return $\mathcal{R} \cup \mathcal{R}_\infty \models \alpha \rightarrow \beta$
(4) $\mathcal{R} = \mathcal{R} \setminus \mathcal{R}_i$
(5) if $\mathcal{R} \equiv \emptyset$
    (a) return $\mathcal{R}_\infty \models \alpha \rightarrow \beta$
(6) $i = i + 1$
(7) go to (3)
```

This algorithm only returns true if the query is defeasibly entailed in terms of the minimal ranked interpretation for the knowledge base [6].

2.3.4 Example. Given the knowledge base $\mathcal{K} = \{b \mid\!\sim f, p \rightarrow b, p \mid\!\sim \neg f, R \rightarrow b, b \mid\!\sim w\}$, we will use Rational Closure to check if $\mathcal{K} \mid\!\approx p \rightarrow b$.

The result of Base Rank for $\mathcal{K}$ is:

0	$b \rightarrow f, b \rightarrow w$
1	$p \rightarrow \neg f$
∞	$p \rightarrow b, R \rightarrow b$

We start by checking whether $\mathcal{R} \cup \mathcal{R}_\infty \models \neg p$. This is the case as there is no model u of $\overrightarrow{\mathcal{K}}$ such that $u \Vdash p$. Because of this, we remove the most preferred rank from $\mathcal{R}$, i.e., $\mathcal{R}_0$, producing:

0	~~$b \rightarrow f, b \rightarrow w$~~
1	$p \rightarrow \neg f$
∞	$p \rightarrow b, R \rightarrow b$

We check again whether $\mathcal{R} \cup \mathcal{R}_\infty \models \neg p$ and find that it does not. We now check whether $\mathcal{R} \models p \rightarrow b$ which it clearly does since $p \rightarrow b$ is a formula within $\mathcal{R}_\infty$. Thus, according to Rational Closure, $\mathcal{K} \mid\!\approx p \mid\!\sim b$.

3 AIMS AND EXPERIMENT DESIGN

3.1 Aims

The primary aims of this paper are twofold.

Firstly, to develop a knowledge base generation tool that can be used for knowledge base creation and parameterized generation processes, as well as allow for future expansions of other knowledge base generation methodologies and new types of propositional formulas, connectives and operators. Efforts will also be made to allow for the easy integration of the knowledge base generation tool into a larger framework, or simply to be used with ease from the command-line.

Secondly, to investigate and design methodologies for parameterized knowledge base generation and identify the extent to which the parameters identified are useful in the testing of scalability and robustness with respect to two propositional defeasible reasoning implementations and their optimizations, them being Rational Closure, presented in Hamilton's paper, and Lexicographic Closure, presented in Park's paper.

3.2 Experiment Design

Firstly, a knowledge base generation tool will be developed. Attempts will be made to maximize inheritance and polymorphism, allowing for easy future expansion. The success of the knowledge base generation tool will be judged qualitatively, whereby the ease of expansion and integration will be analysed through the system structure and comments from Hamilton and Park will be used to analyse the ease of use of the knowledge base generation tool while testing.

Secondly, methodologies for parameterized knowledge base generation will be designed and implemented. Four parameters for certain will be implemented, i.e., the total number of defeasible ranks, the total number of defeasible statements, the distribution of the defeasible statements among the defeasible ranks, and whether or not the knowledge base should contain classical statements. After Hamilton and Park have completed their testing and analysis using the knowledge base generation tool, a qualitative analysis be performed on their findings, identifying which parameters led to the identification of interesting aspects of their implementations and optimizations, and hence, which were useful.

4 KNOWLEDGE BASE GENERATION

4.1 Simple Implication Knowledge Bases

A knowledge base that results in two defeasible ranks can be easily expressed using three simple implications.

$$\mathcal{K} = \{\mathsf{b} \rightarrow \mathsf{a}, \mathsf{a} \mid\!\sim \mathsf{c}, \mathsf{b} \mid\!\sim \neg\mathsf{c}\}$$

For the purpose of generating more ranks, we can encapsulate the structure of this knowledge base using a graph, i.e.,

$$\begin{tikzcd} \mathsf{a} & \mathsf{b} \arrow[l] \end{tikzcd}$$

where the arrow represents the three statements found in $\mathcal{K}$, and will be referred to as a directed *clash relation* from b to a. Since only the fact this results in an additional rank is of concern, for rank generation purposes c atom need only be a new atom introduced into the knowledge base. Thus, antecedent a has a maximum clash path of depth 1, and b a of depth 1, depicting their residing rank. It should be noted, $\mathsf{b} \rightarrow \mathsf{a}$ can be replaced with $\mathsf{b} \mid\!\sim \mathsf{a}$, and the same structure will hold. We can therefore represent a knowledge with two ranks using the graph

$$\begin{tikzcd} \mathsf{a} & \mathsf{b} \arrow[l] & \mathsf{c} \arrow[l] \end{tikzcd}$$

which could resolve to the knowledge base

$$\mathcal{K} = \{\mathsf{b} \rightarrow \mathsf{a}, \mathsf{a} \mid\!\sim \mathsf{d}, \mathsf{b} \mid\!\sim \neg\mathsf{d}, \mathsf{c} \rightarrow \mathsf{b}, \mathsf{b} \mid\!\sim \mathsf{e}, \mathsf{c} \mid\!\sim \neg\mathsf{e}\}$$

which after Base Rank will result in three ranks.

If we wish to add statements to a defeasible rank, we can simply add a new defeasible statement directed to one of that ranks antecedent. For example, adding the defeasible statement $\mathsf{f} \mid\!\sim \mathsf{c}$ to $\mathcal{K}$ will result in the statement $\mathsf{f} \mid\!\sim \mathsf{c}$ being added to the rank 2.

We can express this using the graph

$$\begin{tikzcd} \mathsf{a} & \mathsf{b} \arrow[l] & \mathsf{c} \arrow[l] & \mathsf{f} \arrow[l, dotted] \end{tikzcd}$$

where the dotted arrow represents a directed *defeasible relation* from f to c. Antecedents that have outgoing defeasible relations transitively assume their consequent's position in clash paths, hence f has maximum clash path depth of 2.

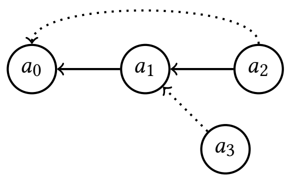
4.2 Defeasible Clash Graphs

4.2.1 Definition. A defeasible clash graph (DCG) is defined as the ordered 3-tuple

$$\mathcal{G} = (\mathcal{A}, \mathcal{C}, \mathcal{D})$$

where $\mathcal{A}$ is a set of antecedent nodes, $\mathcal{C}$ is a set of directed clash relations and $\mathcal{D}$ is a set of directed defeasible relations.

For example, consider the DCG $\mathcal{G}_1 = (\mathcal{A}, C, \mathcal{D})$ where $\mathcal{A} = \{a_0, a_1, a_2, a_3\}$, $C = \{(a_1, a_0), (a_2, a_1)\}$ and $\mathcal{D} = \{(a_2, a_0), (a_3, a_1)\}$.



$\mathcal{G}_1 = \{\{a_0, a_1, a_2, a_3\}, \{(a_1, a_0), (a_2, a_1)\}, \{(a_2, a_0), (a_3, a_1)\}\}$

We will refer to the first and second element of a directed relation as the antecedent and the consequent of the relation respectively. An antecedents may be involved in many clash path of which its maximum clash path depth depicts its resulting defeasible rank. For example, a_2 is part of a 0 depth clash path owing to its defeasible relation with a_0, and a clash path of depth 2 owing to its clash relation with a_1, hence a_2 has a maximum clash path depth of 2, signifying its defeasible rank postion. Cyclical clash path's force all their members, as well as the ones they associate towards, to the infinite rank.

4.2.2 Conversion to Knowledge Base. DCGs represent the underlying conflict structure among antecedents in a knowledge base rather than the actual formulas present in the knowledge base itself. What follows is a description of how knowledge bases can be generated from DCGs.

A defeasible relation represents a simple directed association from an antecedent to a consequent and is added directly as a single defeasible implication into the knowledge base. A clash relation represents both a directed association, which can be either classical or defeasible, and a pair of defeasible associations between the antecedent and the consequent that form a contradiction and as such, three additional formulas are added to the knowledge base:

(1) A defeasible/classical implication associating the antecedent atom with the consequent atom.
(2) A defeasible implication associating the antecedent atom with a new contradiction atom.
(3) A defeasible implication associating the consequent atom with the negation of the new contradiction atom.

The *DCGKB* algorithm (see algorithm 1) converts a DCG into a knowledge base containing either classical and defeasible implications, or just defeasible implications. It's important to note, the classical formula $\alpha \rightarrow \beta$ is equivalent to the defeasible formula $\neg(\alpha \rightarrow \beta) \mid\sim \bot$.

As an example, consider $\mathcal{G}_1$ as defined above, with $defeasibleOnly$ set to true. Using the DCGKB algorithm, the following knowledge base is produced:

$$\begin{aligned}\mathcal{K}_1 = \{&a_1 \mid\sim a_0, a_1 \mid\sim \gamma_0, a_0 \mid\sim \neg\gamma_0,\\ &a_2 \mid\sim a_1, a_2 \mid\sim \gamma_1, a_1 \mid\sim \neg\gamma_1,\\ &a_3 \mid\sim a_1,\\ &a_2 \mid\sim a_0\}\end{aligned}$$

Algorithm 1 DCGKB

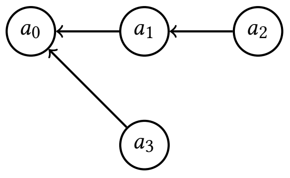
```
1: INPUT: A defeasible clash graph $(\mathcal{A}, C, \mathcal{D})$ and a defeasible
   only Boolean $defeasibleOnly$
2: OUTPUT: A defeasible knowledge base $\mathcal{K}$
3: $\mathcal{K} := \emptyset$
4: $i := 0$
5: for all $(\alpha, \beta) \in C$ do
6:     if $defeasibleOnly$ then
7:         $\mathcal{K} := \mathcal{K} \cup \{\alpha \mid\sim \beta\}$
8:     else
9:         $\mathcal{K} := \mathcal{K} \cup \{\alpha \rightarrow \beta\}$
10:    $\mathcal{K} := \mathcal{K} \cup \{\alpha \mid\sim \gamma_i\}$
11:    $\mathcal{K} := \mathcal{K} \cup \{\beta \mid\sim \neg\gamma_i\}$
12:    $i := i + 1$
13: for all $(\alpha, \beta) \in \mathcal{D}$ do
14:    $K := K \cup \{\alpha \mid\sim \beta\}$
15: return $\mathcal{K}$
```

4.2.3 DCG Simplication. Defeasible clash graphs containing defeasible relations can be simplified to only containing clash relations. While this inherently loses information about the initial graph, it retains an antecedent's position in clash paths and simplifies the process of predicting antecedent ranks. The *SimplifyDCG* algorithm (see algorithm 2) performs this operation.

First, Warshall's algorithm is computed on the digraph $(\mathcal{A}, \mathcal{D})$, and the transitive defeasible relations are extracted. Clash relations are then added associating antecedents with the consequents of any clash relation of a consequent that the antecedent was a direct or transitive antecedent of. When applying SimplifyDCG to $\mathcal{G}_1$, the following DDG is produced:

Algorithm 2 SimplifyDCG

```
1: INPUT: A defeasible clash graph $(\mathcal{A}, C_0, \mathcal{D})$
2: OUTPUT: A simple defeasible clash graph $(\mathcal{A}, C_1, \emptyset)$
3: $\mathcal{D}$ := Warshall($\mathcal{D}$)
4: $C_1 := C_0$
5: for all $(\alpha, \beta) \in \mathcal{D}$ do
6:     for all $(\gamma, \delta) \in C_0$ do
7:         if $\gamma = \beta$ then
8:             $C_1 := C_1 \cup (\alpha, \delta)$
9: return $(\mathcal{A}, C_1, \emptyset)$
```



$\mathcal{G}_2$

4.2.4 Predicting Antecedent Rankings. In order to predict the resulting defeasible rank of each antecedent, the CompatibilityClosure (see algorithm 3) can be used, wherein a variation of the Floyd-Warshall algorithm is implemented in order to find an antecedent's longest clash path.

Algorithm 3 CompatibilityClosure

```
INPUT: A simple defeasible clash graph $(\mathcal{A}, C)$
OUTPUT: A compatibility set $E$
Compatibility matrix $\mathcal{M} := |A| \times |A|\ -\infty$ matrix
for all $(\alpha, \beta) \in C$ do
    $\mathcal{M}[\alpha][\beta] := 1$
for all $k \in \mathcal{A}$ do
    for all $j \in \mathcal{A}$ do
        for all $i \in \mathcal{A}$ do
            if $\mathcal{M}[j][k] = \infty$ then
                $\mathcal{M}[j][i] = \infty$
            else if $\mathcal{M}[i][k] > 0$ and $\mathcal{M}[||][|] \neq -\infty$ then
                $\mathcal{M}[j][i] = \mathbf{max}(\mathcal{M}[j][i], \mathcal{M}[i][k] + \mathcal{M}[k][j])$
            if $j = i$ and $\mathcal{M}[j][i] \neq -\infty$ then
                $\mathcal{M}[j][i] := \infty$
$E := \emptyset$
for all $\alpha \in \mathcal{A}$ do
    $E := E \cup (\alpha, \mathbf{max}(\mathcal{M}[\alpha], 0))$
return $E$
```

For $\mathcal{G}_2$, the compatibility matrix $\mathcal{M}$ would be

$$\begin{bmatrix} -\infty & -\infty & -\infty & -\infty \\ 1 & -\infty & -\infty & -\infty \\ 2 & 1 & -\infty & -\infty \\ 1 & -\infty & -\infty & -\infty \end{bmatrix}$$

and thus, the compatibility set $E = \{(a_0, 0), (a_1, 1), (a_2, 2), (a_3, 1)\}$.

If the knowledge base $\mathcal{K}_1$ built from $\mathcal{G}_1$ is ranked, the following ranked interpretation is produced:

0	$a_0 \mid\!\sim \neg\gamma_0$
1	$a_1 \mid\!\sim \gamma_0, a_1 \mid\!\sim \neg\gamma_1$
2	$a_2 \mid\!\sim \gamma_1$
∞	$a_2 \rightarrow a_0, a_1 \rightarrow a_0, a_3 \rightarrow a_1, a_2 \rightarrow a_1$

As can be seen from above, each antecedent's rank corresponds to their predicted defeasible rank found in the compatibility matrix.

4.3 Parameterized Generation

For the initial parameterized generation of knowledge bases, three parameters were taken into account, i.e., the defeasible ranks count, the total number of defeasible statements, and whether the knowledge base should only contain defeasible statements. In order to accommodate these, a special case of DCGs is constructed and then converted to a knowledge base.

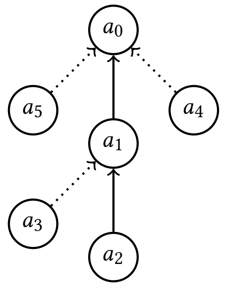
4.3.1 *Chain DCGs.*

Chain DCGs are categorized as DCGs where there is a single linear chain of clash relations, of which singular defeasible implications branch on to.

For example, the DCG

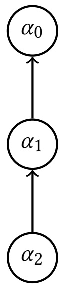
$$\begin{aligned}\mathcal{G}_3 = \{&\{a_0, a_1, a_2, a_3, a_4, a_5\},\\ &\{(a_1, a_0), (a_2, a_1)\},\\ &\{(a_5, a_0), (a_4, a_0), (a_3, a_1)\}\}\end{aligned}$$

is a Chain DCG, and has the graphical form:



$\mathcal{G}_3$

Using Chain DCGs for knowledge base generation with classical statements forces a lower bound of 2 statements per rank. The reason for this is exemplified by the simple chain DCG, $\mathcal{G}_4 = \{\{\alpha_0, \alpha_1, \alpha_2\}, \{(\alpha_1, \alpha_0), (\alpha_2, \alpha_1)\}, \emptyset\}$.



$\mathcal{G}_4$

If we apply DCGKB with $defeasibleOnly$ set to false $\mathcal{G}_4$, and then Base Rank to the result, the following result is produced

0	$a_0 \mid\!\sim \gamma_0$
1	$a_1 \mid\!\sim \neg\gamma_0, a_1 \mid\!\sim \gamma_1$
2	$a_2 \mid\!\sim \neg\gamma_1$
∞	$a_1 \rightarrow a_0, a_2 \rightarrow a_2$

As can be seen, rank 1 contains 2 formulas. As such, each defeasible rank is forced to have 2 formulas for the sake of reliability. Processing through DCGKB with $defeasibleOnly$ set to true adds an additional formula to each defeasible rank, for a lower bound of 3 per rank.

4.3.2 *Statement Distributions.*

Five distributions were implemented, i.e., uniform, exponential, normal, inverted-exponential and inverted-normal, for distributing defeasible statements over ranks and providing knowledge bases adhering to different shapes.

Cumulative distribution functions are used to calculate the statements to be placed in a specific rank. Each cumulative distribution was fit so the y axis correlated to the total number of statements in the defeasible ranks, with the x axis corresponding to the total number at that rank. To find how many statements to were to be added to a specific rank, the previous defeasible rank's total value was subtracted from the current defeasible rank's total. In order to optimize the process, the subtraction equation was simplified.

The distribution functions used are described as follows, where x is the current rank, s is the total number of defeasible statements, and r is the total number of defeasible ranks:

$$\begin{aligned}
\textbf{Uniform}(s,r,x) &= \frac{s}{r} \\
\textbf{Exponential}(s,r,x) &= s((1-e^{-x\frac{5}{r}}) - (1-e^{-(x-1)\frac{5}{r}})) \\
&= s(e^{-\frac{5(x-1)}{r}+1} - e^{-\frac{x5}{r}}) \\
\textbf{Normal}(s,r,x) &= s\left(\frac{\sqrt{\frac{\pi}{2}}\,\mathbf{erf}(-\frac{4}{2}+\frac{8x}{2r})+\sqrt{\frac{\pi}{2}}}{\sqrt{2}\sqrt{\pi}} - \frac{\sqrt{\frac{\pi}{2}}\,\mathbf{erf}(-\frac{4}{2}+\frac{8(x-1)}{2r})+\sqrt{\frac{\pi}{2}}}{\sqrt{2}\sqrt{\pi}}\right) \\
&= s\left(\frac{\mathbf{erf}(\frac{4x}{r}-2)-\mathbf{erf}(\frac{4(x-1)}{r}-2)}{2}\right) \\
\textbf{InvertedNormal}(s,r,x) &= \begin{cases}\textbf{Normal}(s,r,x+\frac{r+1}{2}) & \text{if } x < \frac{r+1}{2} \\ \textbf{Normal}(s,r,x-\frac{r+1}{2}) & \text{otherwise}\end{cases} \\
\textbf{InvertedExponential}(s,r,x) &= s\left(e^{-\frac{5((r+1-x)-1)}{r}+1} - e^{-\frac{(r-x)5}{r}}\right)
\end{aligned}$$

Algorithm 4 RankedGenerate

```
INPUT: A number of defeasible ranks $r$, total number of defeasible statements $s$, use only defeasible Boolean $defeasibleOnly$ and a distribution function $d$
OUTPUT: A defeasible knowledge base $\mathcal{K}$
$\mathcal{A} := \emptyset$
$\mathcal{C} := \emptyset$
$\mathcal{D} := \emptyset$
if $defeasibleOnly$ then
    $lowerBound := 3$
else
    $lowerBound := 2$
$j := 0$
while $j < r$ do
    $\mathcal{A} := \mathcal{A} \cup \{\alpha_j\}$
    if $j > 0$ then
        if defeasibleOnly then
            $\mathcal{D} := \mathcal{D} \cup \{(\alpha_j, \alpha_{j-1})\}$
        else
            $\mathcal{C} := \mathcal{C} \cup \{(\alpha_j, \alpha_{j-1})\}$
    if r = 1 then
        $i := 0$
    else if $j = r - 1$ then
        $i := 1$
    else
        $i := 2$
    while $i < \mathbf{max}(\mathbf{d}(s, r, j+1), lowerBound)$ do
        $\mathcal{A} := \mathcal{A} \cup \{\delta_{j,i}\}$
        $\mathcal{D} := \mathcal{D} \cup \{(\delta_{j,i}, \alpha_j)\}$
        $i := i + 1$
    $j := j + 1$
$\mathcal{K} := \mathbf{DCGKB}((\mathcal{A}, \mathcal{C}, \mathcal{D}), defeasibleOnly)$
return $\mathcal{K}$
```

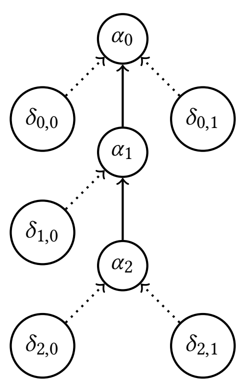
4.3.3 *Basic Knowledge Base Generation.*

The RankedGenerate algorithm (see algorithm 4) generates a knowledge base according to a total number of defeasible ranks, a total number of defeasible statements, a use only defeasible Boolean, and a distribution function.

For example, given the parameters $r = 3$, $s = 9$ and $d = uniform$, RankedGenerate produces the following chain DCG

$$\begin{aligned}
\mathcal{G}_5 = \{&\{\alpha_0, \alpha_1, \alpha_2, \delta_{0,0}, \delta_{0,1}, \delta_{1,0}, \delta_{1,1}, \delta_{2,0}, \delta_{2,1}\},\\
&\{(\alpha_1, \alpha_0), (\alpha_2, \alpha_1)\},\\
&\{(\delta_{0,0}, \alpha_0), (\delta_{0,1}, \alpha_0), (\delta_{1,0}, \alpha_1), (\delta_{2,0}, \alpha_2), (\delta_{2,1}, \alpha_2)\}\}
\end{aligned}$$

with the graphical representation



$\mathcal{G}_5$

producing the knowledge base

$$\begin{aligned}\mathcal{K}_5 = \{&\alpha_1 \rightarrow \alpha_0, \alpha_1 \mathrel|\!\sim \neg\gamma_0, \alpha_0 \mathrel|\!\sim \gamma_0,\\ &\alpha_2 \rightarrow \alpha_1, \alpha_2 \mathrel|\!\sim \neg\gamma_1, \alpha_2 \mathrel|\!\sim \gamma_1,\\ &\delta_{0,0} \mathrel|\!\sim \alpha_0, \delta_{0,1} \mathrel|\!\sim \alpha_0,\\ &\delta_{1,0} \mathrel|\!\sim \alpha_1,\\ &\delta_{2,0} \mathrel|\!\sim \alpha_2, \delta_{2,1} \mathrel|\!\sim \alpha_2\}\end{aligned}$$

and the ranking of $\mathcal{K}_5$

0	$\delta_{0,0} \mathrel\mid\!\sim \alpha_0, \delta_{0,1} \mathrel\mid\!\sim \alpha_0, \alpha_0 \mathrel\mid\!\sim \gamma_0$
1	$\delta_{1,0} \mathrel\mid\!\sim \alpha_1, \alpha_1 \mathrel\mid\!\sim \neg\gamma_0, \alpha_1 \mathrel\mid\!\sim \gamma_1$
2	$\delta_{2,0} \mathrel\mid\!\sim \alpha_2, \delta_{2,1} \mathrel\mid\!\sim \alpha_2, \alpha_2 \mathrel\mid\!\sim \neg\gamma_1$
∞	$\alpha_1 \rightarrow \alpha_0, \alpha_2 \rightarrow \alpha_1$

4.3.4 Conservative Knowledge Base Generation. If $defeasibleOnly$ is set to false, the lower-bound of the number of defeasible statements per rank can be reduced to 1 if a conservative method of knowledge base conversion is used. If only defeasible statements are generated, this bound raises to 2. This can be accomplished by using a single atom for clash relations, rather than generating a new one each time. The ChainDCGKB algorithm (see algorithm 5) uses this tactic to produce a knowledge base from a Chain DCG. If we apply ChainDCGKB to $\mathcal{G}_4$, we get the following knowledge

Algorithm 5 ChainDCGKB

```
1: INPUT: A chain defeasible clash graph $(\mathcal{A}, \mathcal{C}, \mathcal{D})$ and defeasible only Boolean $defeasibleOnly$
2: OUTPUT: A defeasible knowledge base $\mathcal{K}$
3: $\mathcal{K} := \emptyset$
4: $j := 0$
5: $\mathcal{B} := \mathcal{A} \setminus \{\alpha | (\alpha, \beta) \in \mathcal{D}\}$
6: $(\alpha_0, \ldots, \alpha_{|\mathcal{B}|-1}) :=$ TopologicalSort$(\mathcal{B}, \mathcal{C})$
7: while $j < |\mathcal{B}|$ do
8:     if $j > 0$ then
9:         if $defeasibleOnly$ then
10:            $\mathcal{K} := \mathcal{K} \cup \{\alpha_j \mid\!\sim \alpha_{j+1}\}$
11:        else
12:            $\mathcal{K} := \mathcal{K} \cup \{\alpha_j \rightarrow \alpha_{j+1}\}$
13:    if $j + 1\%2 = 0$ then
14:        $\mathcal{K} := \mathcal{K} \cup \{\alpha_j \mid\!\sim \gamma\}$
15:    else
16:        $\mathcal{K} := \mathcal{K} \cup \{\alpha_j \mid\!\sim \neg\gamma\}$
17:    $j := j + 1$
18: for all $(\alpha, \beta) \in \mathcal{D}$ do
19:     $K := K \cup \{\alpha \mid\!\sim \beta\}$
20: return $\mathcal{K}$
```

base

$$\begin{aligned}\mathcal{K}'_4 = \{&\alpha_0 \mathrel|\!\sim \gamma,\\ &\alpha_1 \rightarrow \alpha_0, \alpha_1 \mathrel|\!\sim \neg\gamma\\ &\alpha_2 \rightarrow \alpha_1, \alpha_2 \mathrel|\!\sim \gamma\}\end{aligned}$$

leading to the ranked knowledge base

0	$\alpha_0 \mathrel\mid\!\sim \gamma$
1	$\alpha_1 \mathrel\mid\!\sim \neg\gamma$
2	$\alpha_2 \mathrel\mid\!\sim \gamma$
∞	$\alpha_1 \rightarrow \alpha_0, \alpha_2 \rightarrow \alpha_1$

Thus, the ConservativeRankedGenerate algorithm (see algorithm 6) makes use of this method.

Algorithm 6 ConservativeRankedGenerate

```
1: INPUT: A number of ranks $r$, total number of statements $s$, defeasible only Boolean $defeasibleOnly$ and a distribution function $d$
2: OUTPUT: A defeasible knowledge base $\mathcal{K}$
3: $\mathcal{A} := \emptyset$
4: $\mathcal{C} := \emptyset$
5: $\mathcal{D} := \emptyset$
6: if defeasibleOnly then
7:     $lowerBound := 2$
8: else
9:     $lowerBound := 1$
10: $j := 0$
11: while $j < r$ do
12:     $\mathcal{A} := \mathcal{A} \cup \{\alpha_j\}$
13:     if $j > 0$ then
14:         $\mathcal{C} := \mathcal{C} \cup \{(\alpha_j, \alpha_{j-1})\}$
15:     if $r = 1$ or $j = 0$ then
16:         $i := 1$
17:     else
18:         $i := 2$
19:     while $i <$ max(d$(s, r, j + 1), lowerBound)$ do
20:         $\mathcal{A} := \mathcal{A} \cup \{\delta_{j,i}\}$
21:         $\mathcal{D} := \mathcal{D} \cup \{(\delta_{j,i}, \alpha_j)\}$
22:         $i := i + 1$
23:     $j := j + 1$
24: $\mathcal{K} :=$ ConservativeDCGKB$((\mathcal{A}, \mathcal{C}, \mathcal{D}), defeasibleOnly)$
25: return $\mathcal{K}$
```

5 DESIGN AND IMPLEMENTATION

The knowledge base generation tool (KBGT) is a system for knowledge base ranking, storage, creation, and generation.

5.1 Environment

The system was developed using Scala 2.13.6. Scala provides a good balance between the imperative and functional paradigms which compliments a propositional logic implementation, allowing for concise and robust data-structures to be developed, along with declarative function sequences that promote compiler optimization. Scala compiles to JVM bytecode, allowing for the easy integration and usage of KBGT functionality in JVM based projects. This also allows good levels of portability.

5.2 Architecture

KBGT is composed of two core modules. The *logic* library contains classical and defeasible propositional logic types and functionality

The *generation* library contains knowledge base generation related types and functionality.

5.2.1 Class descriptions. Class descriptions for both modules can be found in appendix A.

5.2.2 Data-structures. Classical formulas are stored as recursively constructed trees, which are space efficient. Since the focus of KBGT is on knowledge base manipulation, new classical formulas are constantly constructed, meaning the fact these trees are immutable is ideal, as they are stored, operated upon, and moved around efficiently. Defeasible formulas store a 2-tuple of these trees, i.e., the antecedent and consequent of the defeasible implication, and is also immutable. Both classical and defeasible formulas can therefore be easily processed through recursive methods that parse their respective trees. All knowledge bases inherit from the mutable Set type, which has an effectively constant time for all used operations. This allows formulas to be quickly added and removed, as well as retain the typical set properties of a knowledge base and extend typical knowledge base functionality through inherited methods. The ranked knowledge base utilizes a ListBuffer for the storage of defeasible ranks. This is one of the most efficient mutable sequential data structures in Scala and appends and removes elements in constant time.

5.2.3 UML Diagram. While it is tricky to design UML diagrams for Scala based projects, an attempt can be found in appendix B.

5.3 Features

5.3.1 TweetyProject Integration. The TweetyProject [13] is a collection of Java libraries pertaining to various forms of KRR logics. KBGT integrates the TweetyProject's propositional logic library. This provides access to the TweetyProject's data-structures, for which conversion methods are provided by KBGT from KBGT propositional logic data-structures to their respective TweetyProject counterparts. This is particularly attractive, since along with many other mathematical optimizations for SAT-solving, the TweetyProject provides an interface to the Sat4j satisfiability solver, the fastest openly available Java based SAT-solver. This means all SAT-solver based problems such as satisfiability solving and classical entailment is done using top-class procedures. A system integrating KBGT can also make use of any KBGT data-structure directly as its TweetyProject counterpart, meaning any system that already uses the TweetyProject as a basis could seamlessly integrate with KBGT. For example, a ClassicalFormula can be easily converted into a TweetyProject PlFormula, a ClassicalKnowledgeBase to a PlBeliefSet, and a RankedKnowledgeBase to a ArrayList[PlBeliefSet].

5.3.2 Parsing/Exporting. KBGT makes use of TweetyProject's propositional parser in order to read and write its various knowledge base types to json files. For ClassicalKnowledgeBases, a simple json array containing the parse strings of each ClassicalFormula is written. DefeasibleKnowledgeBases are written similarly, using a "~>" to represent a defeasible implication. It should be noted that one can first materialize a DefeasibleKnowledgeBase and then write it to a file in order to have pure propositional implications instead. RankedKnowledgeBases are written as a nested json arrays, each representing a rank, the last of which being the infinite rank.

5.3.3 Agnostic Boolean Operators. Boolean operators are stored as Scala Enumerations. It is through these enumerations that an operator's notation, parse notation, and ClassicalFormulas' PlFormula counterpart are defined. Since KBGT SAT-solving is purely based on TweetyProject's data-structures, i.e., PlFormula and PlBeliefSet, the addition of operators is simple, as one would need only define a new value in the enumeration, add the notations and then define a PlFormula that is logically equivalent to the new operator. New connectives can also easily be defined, though it requires adding them inheriting from ClassicalFormula along with defining operators using the above instructions.

5.3.4 Command-line Interface. KBGT has an implemented command-line interface. A user can either execute generation methods directly using command-line arguments and have the resounding knowledge base either output to the screen or written to a specified file, or take part in an interactive knowledge base creation process. The interactive process involves the iterative manipulation of a single knowledge base. Users are able to add and remove formulas manually, read in knowledge base files (classical, defeasible, mixed or ranked), write statements from the current knowledge base to a file (also classical, defeasible, mixed or ranked), generate a new knowledge base using the methods previously described in this paper, and finally, view the result of base ranking the knowledge base.

6 ANALYSIS

Five parameters were established, i.e., the total number of defeasible statements, the defeasible rank count, the distribution of the defeasible statements over the defeasible ranks, whether conservative generation should be used, and if the knowledge base should contain defeasible statements only. Knowledge bases generated only contained simple implications of the form $\alpha \rightarrow \beta$, $\alpha \mid\sim \beta$, and $\alpha \mid\sim \neg\beta$, where α and β are propositional atoms.

6.1 Knowledge Base Generation Tool

6.1.1 Ease of expansion. KBGT makes heavy use of inheritance and as such, new types of propositional formulas and knowledge bases can be introduced with readily available inherited functionality.

In KBGT, propositional connectives are decoupled from their respective Boolean operators. This means additional operators can be implemented seamlessly. Along with this, new n-arity connectives can also be added, along with their respective Boolean operators.

The strong distinction between KBGT's data-structures and the knowledge base generation methods allows for the addition of new knowledge base generation methods whilst utilizing the same data-structures.

6.1.2 Ease of integration. All knowledge base generation methods present in KBGT are static, meaning their use by systems that integrate KBGT is unobstructed by instance requirements.

All propositional data-structures in KBGT can be easily converted into TweetyProject's propositional logic data-structures, meaning that any reasoning system that heavily relies on TweetyProject can be easy use of KBGT's methods and data-structures.

6.1.3 Ease of use. There are two paths to accessing KBGT's knowledge base generation facilities. Either the generation is executing directly through the command-line with the generation parameters taken in as command-line arguments with the generated knowledge base being output to terminal or json file, or through an interactive REPL that allows the user to generate new knowledge bases, load/write knowledge bases to and from files, manipulate knowledge bases through the addition and subtraction of defeasible/classical propositional formulas, and view the Base Rank of the current state of the knowledge base.

The fact that KBGT uses a typical json format means that a knowledge bases produce and written to a files do not necessarily have to be parsed by KBGT, but rather by a system that integrates the TweetyProject's propositional logic parser.

6.2 Rational Closure Testing

For Rational Closure, defeasible only knowledge bases using conservative generation were tested upon.

Two sets of tests were done for each optimization.

1. The number of defeasible statements set to 2 with the number of ranks varied between 10, 50 and 100.
2. The number of defeasible ranks set to 50, total number of defeasible statements set to 500, and the distribution function varied between uniform, exponential and normal.

For each of these sets, five different query sets were used.

1. A set of queries whose antecedents all differ from one another.
2. A set of queries which all have the same antecedent.
3. A set of queries whose antecedents are half unique, and are half repeated.
4. A set of queries whose antecedents all become consistent with the knowledge base after all but the final rank have been removed.
5. A set of queries whose antecedents all become consistent with knowledge base after the first rank has been removed.

6.2.1 Defeasible Rank Count. The testing process for Rational Closure revealed that queries whose antecedents became compatible with the knowledge base in the final rank were computed quicker by the naive Rational Closure implementation for knowledge bases with less ranks, but that those queries which become compatible with the knowledge base in the first rank did not change with varying numbers of ranks. This suggests that it is not the number of defeasible ranks which directly impacts the performance of the entailment-checking aspect of the Rational Closure algorithm, but instead it is the rank number at which queries' antecedents become compatible with the knowledge base which is the stronger determinant of the performance of this algorithm. This is a useful insight provided by the defeasible rank count parameter and thus depicts its usefulness.

6.2.2 Defeasible Statement Count. Increasing the total number of defeasible statements did not affect the operation of Rational Closure, which points to the aspect that the same entailment-checking operation is performed no matter the number of statements. This is a useful insight gained from varying the defeasible statement count.

6.2.3 Distribution Function. As stated above, the number of statements per rank does not effect the operation of Rational Closure, which means the distribution function is not a useful parameter for testing, but it could be a useful parameter for analysis.

6.2.4 Defeasible Only. Defeasible only was always true for the Rational Closure evaluation and as such the usefulness of defeasible only parameter cannot be spoken for.

6.2.5 Conservative Generation. Conservative generation was only ever utilized and therefore the usefulness of the regular generation compared to conservative generation cannot be spoken for.

6.3 Lexicographic Closure Testing

For Lexicographic Closure, separate tests were conducted for both defeasible only knowledge bases and knowledge bases containing both defeasible and classical statements.

For knowledge bases containing both classical and defeasible statements, a normal distribution was selected with 200 total defeasible statements and 49 defeasible ranks where queries were selected from the knowledge base going up in multiples of their index.

For defeasible only knowledge bases, 50 defeasible ranks with 200 total defeasible statements were kept constant, with the distribution varying between uniform, normal, and inverse-normal. Every defeasible statement within the knowledge base was queried for these tests.

6.3.1 Defeasible Rank Count. The defeasible rank count remained static and thus the usefulness of this parameter for testing Lexicographic Closure cannot be identified explicitly. However, the defeasible rank count was used to verify the approximate time-complexity of the Binary and Ternary search optimizations, so it was useful in playing a role in the analysis.

6.3.2 Defeasible Statement Count. If the distribution function and number of defeasible ranks stays the same, increasing the defeasible statement count extends Lexicographic Closure's execution time. This means the total defeasible statement count is a useful parameter, as it points to the aspect of Lexicographic Closure that more populated defeasible ranks leads to a more expensive refinement process.

6.3.3 Distribution Function. Similarly to analysis found about, the number of defeasible statements per rank greatly impacted the testing owing to the rank refinement process becoming much more expensive the more statements were present within the rank. It took 13 statements within a rank to cause the implementation to run out of JVM heap space.

6.3.4 Defeasible Only. It was found that Lexicographic Closure can process knowledge bases containing defeasible and classical statements faster than a knowledge base containing only defeasible statements but with the same total number of statements and ranks. So being able to select between defeasible only generation and a mixed generation is a useful parameter, since it helped identify that the distribution between defeasible ranks and the infinite rank is performance factor for Lexicographic Closure

6.3.5 *Conservative Generation.*

Only conservative generation was utilized, so the usefulness of the conservative generation parameter cannot be spoken for.

7 RELATED WORK

Kraus, Lehmann, and Magidor [10] introduced the preferential approach to defeasible reasoning including the addition of the "typically" $\mid\sim$ KLM-style defeasible implication and the KLM properties. Lehmann and Magnidor [12] introduced the Base Rank and Rational Closure algorithms for defeasible propositional entailment. Lehmann [11] introduced the Lexicographic Closure algorithm for defeasible propositional entailment.

8 CONCLUSIONS

The goal behind this paper was to investigate knowledge base generation methods and assess their viability with regards to testing the scalability and robustness of practical implementations of the Rational Closure and Lexicographic Closure defeasible entailment implementations and their optimizations.

The knowledge base generation tool developed (KBGT) was found useful by Hamilton and Park when testing and analysing their respective implementations. The fact a Jar could be presented and knowledge bases could be generated immediately through program arguments and written to a file which they could parse into their implementations made using KBGT simple and efficient. KBGT has also been shown to be a tool ready for expansions. This is shown through the ease at which different types of knowledge bases, propositional formulas, connectives, operators, and notations can be introduced. KBGT has been shown to be easily integrable, through the choice of a JVM based language, static generation methods, and easy conversion into widely used TweetyProject datastructures.

Five parameters were established for testing, i.e., the total number of defeasible statements, the total number of defeasible ranks, the distribution of the defeasible statements, whether or not the knowledge base should be defeasible statements only, and whether or not conservative generation should be used.

For all testing, conservative generation was used as it simply reduced the number of atoms in the knowledge base, while reducing the minimum number of defeasible statements per rank, thus pointing to it being a useless parameter.

The total number of defeasible ranks provided useful insights for both the Lexicographic and Rational Closure implementations. For Rational Closure, defeasible rank count was found to not affect the execution with the more important factor being which rank the defeasible query was compatible with. For Lexicographic Closure, the defeasible rank count provided a means of testing the approximate time complexity of the implementation and optimisations. It can be argued then that the total number of defeasible ranks is a useful parameter.

Differing the total number of defeasible statements again showed no difference in the execution of Rational Closure, while for Lexicographic Closure it was found to drastically increase the execution time owing to the refinement process. This shows that the total number of defeasible statements can be a useful parameter for the right algorithms.

Much like differing the total number of defeasible statements, changing the distribution function had no effect on the operation of Rational Closure and is as such not useful in that regard. For Lexicographic Closure, differing distributions once again put tremendous strain on the refinement process, showing it to be a useful parameter for the right algorithm.

Only defeasible only knowledge bases were used in the testing of Rational Closure so its usefulness cannot be spoken for. For Lexicographic Closure, it was found that if the total number of statements remained static, Lexicographic Closure would operate faster if a larger portion of the statements were found in the classical rank, showing the defeasible only parameter to be useful.

9 FUTURE WORK

No forms of non-determinstic, pseudo-random styles of defeasible knowledge base generation were investigated in this paper and as such are left open to further investigation.

The methods of resolving defeasible clash graphs to defeasible knowledge bases outlined in this paper only result in simple implications while it may be possible to generate more complicated propositional and defeasible formulas, utilizing the clash path lengths in different ways to predict a formula's defeasible rank. More complex formulas are particularly pertinent to Lexicographic Closure, wherein during the testing, a maximum of 1 statement was removed every time on the rank where the antecedent was compatibility.

Moving KBGT to Scala3 would allow the Operator type to take in the arity of the Operator extending it using the new Tuple trait. This would allow the *getPlFormula* method to also be inherited. RankedKnowledgeBases store propositional formulas in their respective defeasible/classical forms. This is so that attempts can be made in the future to add and subtract statements from a RankedKnowledgeBase without needing to repeat the entire Base Rank procedure. Optimizations for the Base Rank algorithm should be explored as this was a timely process for knowledge bases containing over 100 ranks.

REFERENCES

[1] Mordechai Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition.* Springer, 2012.

[2] Zied Bouraoui, Antoine Cornuéjols, Thierry Denœux, Sebastien Destercke, Didier Dubois, Romain Guillaume, João Marques-Silva, Jérôme Mengin, Henri Prade, Steven Schockaert, Mathieu Serrurier, and Christel Vrain. From shallow to deep interactions between knowledge representation, reasoning and machine learning (kay r. amel group), 12 2019.

[3] G. Casini, T. Meyer, K. Moodley, and I. Varzinczak. Towards practical defeasible reasoning for description logics. Jul 2013.

[4] Giovanni Casini, Thomas Meyer, Kodylan Moodley, and Riku Nortjé. Relevant closure: A new form of defeasible reasoning for description logics. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence*, pages 92–106, Cham, 2014. Springer International Publishing.

[5] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. Taking defeasible entailment beyond rational closure. In Francesco Calimeri, Nicola Leone, and Marco Manna, editors, *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings*, volume 11468 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2019.

[6] Michael Freund. Preferential reasoning in the perspective of poole default logic. *Artificial Intelligence*, 98(1):209–235, 1998.

[7] Laura Giordano, Valentina Gliozzi, Nicola Olivetti, and Gian Luca Pozzato. Semantic characterization of rational closure: From propositional logic to description logics. *Artif. Intell.*, 226:1–33, 2015.

[8] Ivan Varzinczak Giovanni Casini, Thomas Meyer. Defeasible entailment: from rational closure to lexicographic closure and beyond. In *17th International*

Workshop on Non-Monotonic Reasoning (NMR), pages 109–118, Arizona, USA, 10 2018.

[9] Adam Kaliski. An overview of klm-style defeasible entailment. Master's thesis, Faculty of Science, University of Cape Town, Rondebosch, Cape Town, 7700, 2020.

[10] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial intelligence*, 44(1-2):167–207, 1990.

[11] Daniel Lehmann. Another perspective on default reasoning. *Annals of Mathematics and Artificial Intelligence*, 15(1):61–82, Mar 1995.

[12] Daniel Lehmann and Menachem Magidor. What does a conditional knowledge base entail? *Artificial Intelligence*, 55(1):1–60, 1992.

[13] Matthias Thimm. Tweety: A comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

Appendix

Appendix A

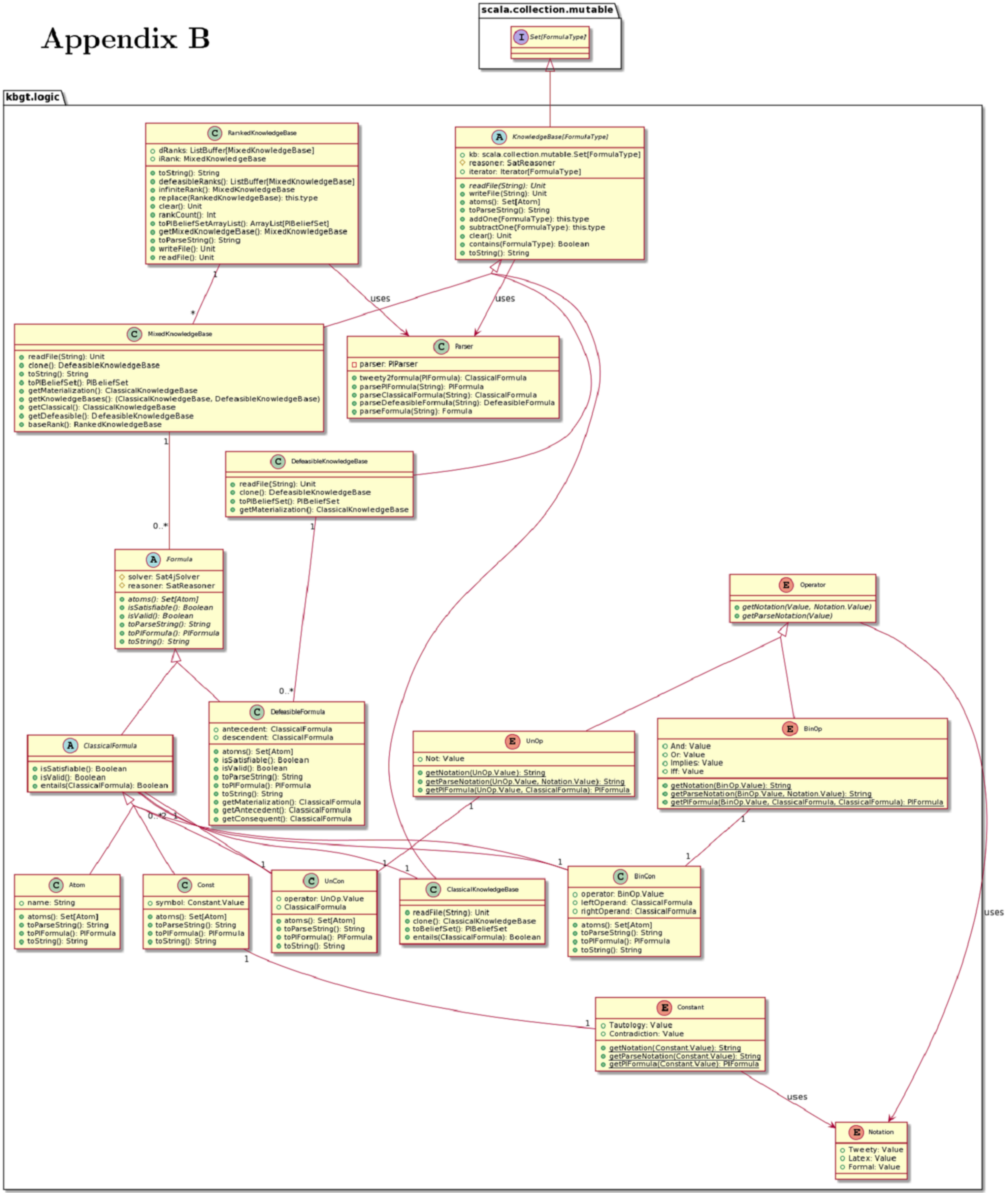
Logic Class Descriptions

- The *Formula* class is an abstract type containing functionality shared by all propositional formulas.
- The *ClassicalFormula* class is the classical sub-type of a propositional formula.
- The *Atom* class is sub-type of a classical formula, representing a propositional atom.
- The *BinCon* class is sub-type of a classical formula, representing a propositional binary connective.
- The *UnCon* class is sub-type of a classical formula, representing a propositional unary connective.
- The *Const* class is sub-type of a classical formula, representing a propositional constant.
- The *DefeasibleFormula* class is the defeasible sub-type of a propositional formula.
- The *Operator* enumeration is a super-type for propositional logic operators.
- The *BinOp* enumeration is an operator sub-type for binary operators.
- The *UnOp* enumeration is an operator sub-type for unary operators.
- The *Constant* enumeration is a symbol enumeration for Tautology and Contradiction constants.
- The *KnowledgeBase* class is an abstract type containing functionality shared by all propositional knowledge bases.
- The *ClassicalKnowledgeBase* class is a classical sub-type of a propositional knowledge base, containing only classical propositional formulas.
- The *DefeasibleKnowledgeBase* class is a defeasible sub-type of a propositional knowledge base, containing only defeasible propositional formulas.
- The *MixedKnowledgeBase* class is a mixed sub-type of a propositional knowledge base, containing both defeasible and classical propositional formulas.
- The *RankedKnowledgeBase* class is a ranked interpretation of a mixed propositional knowledge base.
- The *Parser* class is a parser for propositional formulas.

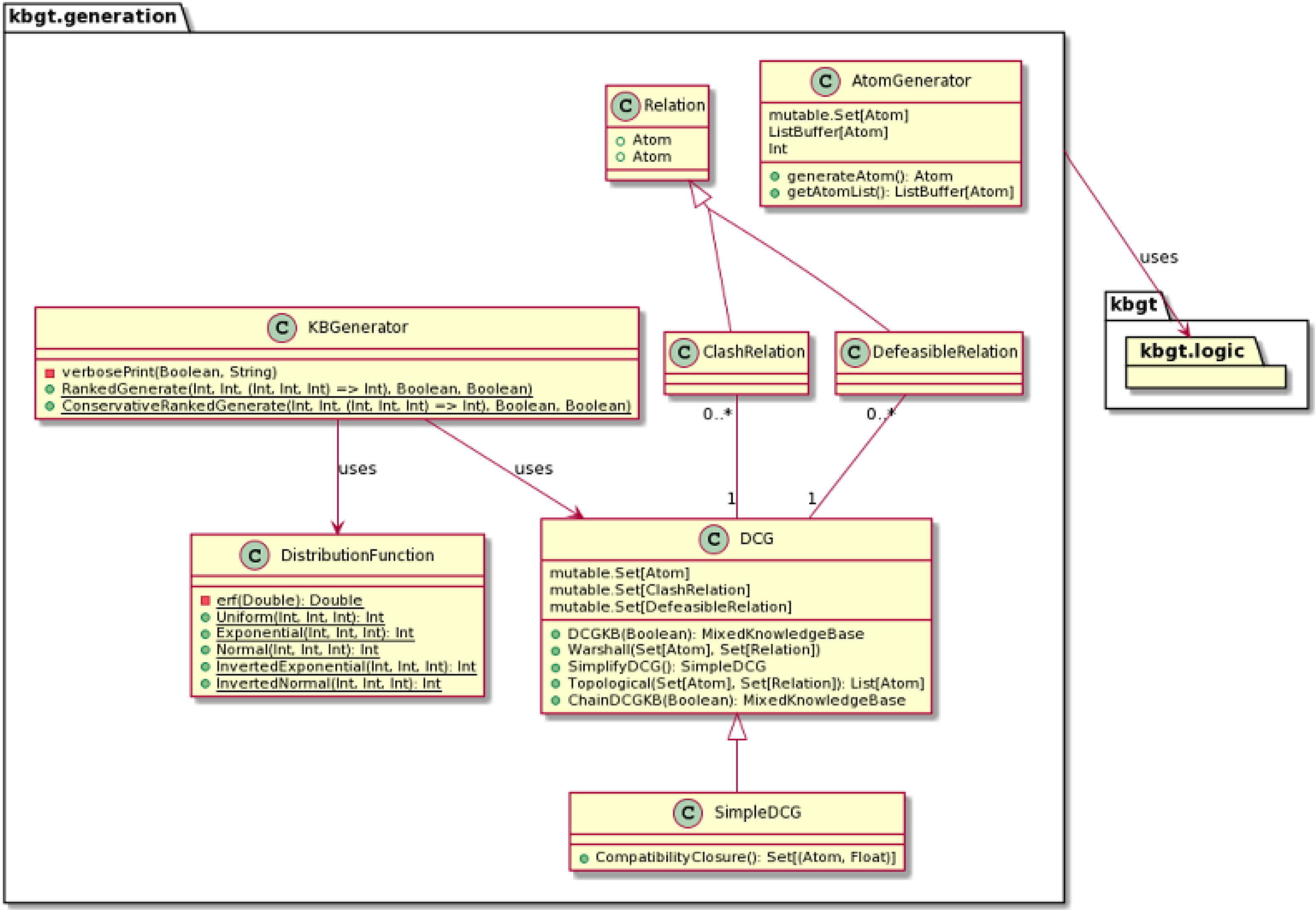
Generation Class Descriptions

- The *AtomGenerator* class is used to generate new atoms given a set of already in-use atoms.
- The *DistributionFunction* class is a collection of distribution functions.
- The *Relation* class contains the types of relation used within defeasible clash graphs.
- The *ClashRelation* class is a clash sub-type of relation.
- The *DefeasibleRelation* class is the defeasible sub-type of relation.
- The *DCG* class is the defeasible clash graph type.
- The *SimpleDCG* class is the simple defeasible clash graph type and inherits from the DCG class.
- The *KBGenerator* class is a collection of methods used to generate knowledge bases using defeasible clash graphs.

Appendix B



scala.collection.mutable
Set[FormulaType]
kbgt.logic
RankedKnowledgeBase
dRanks: ListBuffer[MixedKnowledgeBase]
iRank: MixedKnowledgeBase
toString(): String
defeasibleRanks(): ListBuffer[MixedKnowledgeBase]
infiniteRank(): MixedKnowledgeBase
replace(RankedKnowledgeBase): this.type
clear(): Unit
rankCount(): Int
toPlBeliefSetArrayList(): ArrayList[PlBeliefSet]
getMixedKnowledgeBase(): MixedKnowledgeBase
toParseString(): String
writeFile(): Unit
readFile(): Unit
KnowledgeBase[FormulaType]
kb: scala.collection.mutable.Set[FormulaType]
reasoner: SatReasoner
iterator: Iterator[FormulaType]
readFile(String): Unit
writeFile(String): Unit
atoms(): Set[Atom]
toParseString(): String
addOne(FormulaType): this.type
subtractOne(FormulaType): this.type
clear(): Unit
contains(FormulaType): Boolean
toString(): String
uses
uses
MixedKnowledgeBase
readFile(String): Unit
clone(): DefeasibleKnowledgeBase
toString(): String
toPlBeliefSet(): PlBeliefSet
getMaterialization(): ClassicalKnowledgeBase
getKnowledgeBases(): (ClassicalKnowledgeBase, DefeasibleKnowledgeBase)
getClassical(): ClassicalKnowledgeBase
getDefeasible(): DefeasibleKnowledgeBase
baseRank(): RankedKnowledgeBase
Parser
parser: PlParser
tweety2formula(PlFormula): ClassicalFormula
parsePlFormula(String): PlFormula
parseClassicalFormula(String): ClassicalFormula
parseDefeasibleFormula(String): DefeasibleFormula
parseFormula(String): Formula
DefeasibleKnowledgeBase
readFile(String): Unit
clone(): DefeasibleKnowledgeBase
toPlBeliefSet(): PlBeliefSet
getMaterialization(): ClassicalKnowledgeBase
Formula
solver: Sat4jSolver
reasoner: SatReasoner
atoms(): Set[Atom]
isSatisfiable(): Boolean
isValid(): Boolean
toParseString(): String
toPlFormula(): PlFormula
toString(): String
Operator
getNotation(Value, Notation.Value)
getParseNotation(Value)
DefeasibleFormula
antecedent: ClassicalFormula
descendent: ClassicalFormula
atoms(): Set[Atom]
isSatisfiable(): Boolean
isValid(): Boolean
toParseString(): String
toPlFormula(): PlFormula
toString(): String
getMaterialization(): ClassicalFormula
getAntecedent(): ClassicalFormula
getConsequent(): ClassicalFormula
ClassicalFormula
isSatisfiable(): Boolean
isValid(): Boolean
entails(ClassicalFormula): Boolean
UnOp
Not: Value
getNotation(UnOp.Value): String
getParseNotation(UnOp.Value, Notation.Value): String
getPlFormula(UnOp.Value, ClassicalFormula): PlFormula
BinOp
And: Value
Or: Value
Implies: Value
Iff: Value
getNotation(BinOp.Value): String
getParseNotation(BinOp.Value, Notation.Value): String
getPlFormula(BinOp.Value, ClassicalFormula, ClassicalFormula): PlFormula
Atom
name: String
atoms(): Set[Atom]
toParseString(): String
toPlFormula(): PlFormula
toString(): String
Const
symbol: Constant.Value
atoms(): Set[Atom]
toParseString(): String
toPlFormula(): PlFormula
toString(): String
UnCon
operator: UnOp.Value
ClassicalFormula
atoms(): Set[Atom]
toParseString(): String
toPlFormula(): PlFormula
toString(): String
ClassicalKnowledgeBase
readFile(String): Unit
clone(): ClassicalKnowledgeBase
toBeliefSet(): PlBeliefSet
entails(ClassicalFormula): Boolean
BinCon
operator: BinOp.Value
leftOperand: ClassicalFormula
rightOperand: ClassicalFormula
atoms(): Set[Atom]
toParseString(): String
toPlFormula(): PlFormula
toString(): String
uses
Constant
Tautology: Value
Contradiction: Value
getNotation(Constant.Value): String
getParseNotation(Constant.Value): String
getPlFormula(Constant.Value): PlFormula
uses
Notation
Tweety: Value
Latex: Value
Formal: Value


kbgt.generation
Relation
+ Atom
+ Atom
AtomGenerator
mutable.Set[Atom]
ListBuffer[Atom]
Int
+ generateAtom(): Atom
+ getAtomList(): ListBuffer[Atom]
uses
kbgt
kbgt.logic
KBGenerator
- verbosePrint(Boolean, String)
+ RankedGenerate(Int, Int, (Int, Int, Int) => Int), Boolean, Boolean)
+ ConservativeRankedGenerate(Int, Int, (Int, Int, Int) => Int), Boolean, Boolean)
ClashRelation
DefeasibleRelation
0..*
0..*
uses
uses
1
1
DistributionFunction
- erf(Double): Double
+ Uniform(Int, Int, Int): Int
+ Exponential(Int, Int, Int): Int
+ Normal(Int, Int, Int): Int
+ InvertedExponential(Int, Int, Int): Int
+ InvertedNormal(Int, Int, Int): Int
DCG
mutable.Set[Atom]
mutable.Set[ClashRelation]
mutable.Set[DefeasibleRelation]
+ DCGKB(Boolean): MixedKnowledgeBase
+ Warshall(Set[Atom], Set[Relation])
+ SimplifyDCG(): SimpleDCG
+ Topological(Set[Atom], Set[Relation]): List[Atom]
+ ChainDCGKB(Boolean): MixedKnowledgeBase
SimpleDCG
+ CompatibilityClosure(): Set[(Atom, Float)]