An Integrated Development Environment for Probabilistic Relational Reasoning

MARC FINTHAMMER,
FernUniversität in Hagen, Germany,
marc.finthammer@fernuni-hagen.de

MATTHIAS THIMM,
Technische Universität Dortmund, Germany,
matthias.thimm@tu-dortmund.de

Abstract

This paper presents KReator, a versatile integrated development environment for probabilistic inductive logic programming currently under development. The area of probabilistic inductive logic programming (or statistical relational learning) aims at applying probabilistic methods of inference and learning in relational or first-order representations of knowledge. In the past ten years the community brought forth a lot of proposals to deal with problems in that area which mostly extend existing propositional probabilistic methods like Bayes Nets and Markov Networks on relational settings. Only few developers provide prototypical implementations of their approaches and the existing applications are often difficult to install and to use. Furthermore, due to different languages and frameworks used for the development of different systems the task of comparing various approaches becomes hard and tedious. KReator aims at providing a common and simple interface for representing, reasoning, and learning with different relational probabilistic approaches. It is a general integrated development environment which enables the integration of various frameworks within the area of probabilistic inductive logic programming and statistical relational learning. Currently, KReator implements Bayesian logic programs, Markov logic networks, and relational maximum entropy under grounding semantics. More approaches will be implemented in the near future or can be implemented by researchers themselves as KReator is open-source and available under public license. In this paper, we provide some background on probabilistic inductive logic programming and statistical relational learning and illustrate the usage of KReator on several examples using the three approaches currently implemented in KReator. Furthermore, we give an overview on its system architecture.

Keywords: Probabilistic Reasoning, Relational Representation, Implementation

1 Introduction

Probabilistic inductive logic programming (or *statistical relational learning*) is a very active field in research at the intersection of logic, probability theory, and machine learning, see [4, 3] for some thorough overviews. This area investigates methods for representing probabilistic information in a relational context for both reasoning and learning. Traditionally, the focus on research in probabilistic reasoning has been on propositional models. In the past, a lot of propositional models for probabilistic reasoning have been developed such as Bayes nets and Markov nets [39] or probabilistic conditional logic [43]. But the interest in frameworks and efficient methods for relational probabilistic problems is high since they apply to many applications. The relational structure of many real-world problems (concerning the internet, telecommunication networks, human sciences, bioinformatics, and logistics) as well as the presence of uncertainty in these problems demand sophisticated reasoning and learning methods employing both these concepts [4].

Many researchers developed liftings of propositional probabilistic models to the first-order case in order to take advantage of methods and algorithms already developed. Among these are the well-known Bayesian logic programs [27] and Markov logic networks [8] which extend respectively Bayes nets and Markov networks [39] and are based on knowledge-based model construction [46, 2]. Other approaches also employ Bayes nets for their theoretical foundation like logical Bayesian networks [9] and relational Bayesian networks [21]; or they are influenced by other fields of research like probabilistic relational models [18] by database theory, P-log [1] by answer set programming, and ProbLog [41] by Prolog. There are also some few approaches to apply maximum entropy methods to the relational case [33, 45]. The aforementioned approaches stand representative for a vast variety of different approaches developed in the past ten to twenty years (we refer to [4, 3] for a more elaborate discussion of existing approaches and their history). Although there has been great motivation in developing new approaches for statistical relational learning that deal with specific scenarios, thorough comparisons of approaches are rare. This is no surprise as many approaches build on different logics and employ different methods of propositional probabilistic reasoning methods. Still, there are some exceptions to this statement. In [22] Jaeger introduces *model-theoretic expressivity analysis* in order to compare the expressive power of different approaches to statistical relational learning. There, an approach A is *less or equally expressive* than an approach B if we can find "simple" transformations from the intensional ("the rules") and the extensional ("the signature") parts from A to B, respectively, such that the knowledge modeled in A can be modeled in B as well, see [22] for details. From a theoretical point of view, a classification of the existing approaches in such a manner is definitely desirable and would help to understand the relationships of these approaches more clearly. Unfortunately, in [22] it is only shown that relational Bayesian networks [21] are at least as expressive as Markov logic networks [8]. Furthermore, it is conjectured [22] that Bayesian logic programs [27] are equally expressive as relational Bayesian networks [21]. To our knowledge, no further classifications using model-theoretic expressivity analysis have been made so far. But a similar approach is pursued in [35] where it has been shown that Bayesian logic programs [27] and an extension of stochastic logic programs [36] are of equal expressive power. Furthermore, there are some other attempts to compare approaches for statistical relational learning that focus more on comparisons of implementations like [29].

But thorough comparisons are still missing. One explanation for this is the lack of usable and attractive implementations. Although many of the above approaches have been implemented by their developers in a prototypical manner these implementations are often hard to install or hard to use. Thus, it takes a lot of time and effort to become acquainted with a specific implementation and if one wants to perform comparisons of different approaches this time and effort is multiplied. Even the consistent handling of some simple examples becomes an intricate task since one has to deal in parallel with the specific syntax of each implementation. Thus, there is a great need of a unifying system that allows the integration and comparison of different approaches to statistical relational learning and probabilistic inductive logic programming. This is addressed by the KReator IDE [12] which we present in this paper. KReator is part of the ongoing KReate project[1] which aims at developing a common methodology for learning, modelling and inference in a relational probabilistic framework. Currently, the development of KReator is still in a very early stage but already supports Bayesian logic programs, Markov logic networks, and in particular a new approach for using maximum entropy methods in a relational context [33]. KReator aims at providing a common interface for different approaches to statistical relational learning. That way, KReator supports the researcher and knowledge engineer in developing knowledge bases and employing them in a common and easy-to-use fashion. KReator provides abstract interfaces to common structures like knowledge bases and queries and different approaches can be integrated easily by implementing these interfaces. New approaches can also make use of the logical structures (for predicates, constants, atoms, etc.) already implemented in KReator. As said, KReator is still in an early stage of development and more approaches will be supported in the near future. But KReator is available under the GNU General Public License and publicly available under `http://kreator.cs.tu-dortmund.de`, so developers are also encouraged to add support for their approaches on their own. As KReator provides a common methodology to address the functionalities of an approach—in particular KReator provides project management, file handling, and a graphical and a console-based user interface—developers only need to take care of the essential components of their approach and can neglect the tedious tasks of implementing the surrounding infrastructure. This also yields a great advantage for the user of the system as different approaches can be employed using the same interface and the same methods. The present paper is an extension of [12] and we give an overview on the internal architecture of KReator and illustrate its usage.

The rest of this paper is organized as follows. In Section 2 we give an overview on the approaches of statistical relational learning that are currently supported by KReator, i. e. Bayesian logic programs, Markov logic networks, and the relational maximum entropy approach. We go on in Section 3 with presenting the system architecture of KReator and motivate the main design choices. In Section 4 we give a short manual-style overview on the usage of KReator. In Section 5 we illustrate the usage of KReator and the different representations mechanisms by some more examples. In Section 6 we give some hints on future work and conclude.

2 Relational Probabilistic Knowledge Representation

In the following we give a brief overview on frameworks for relational probabilistic reasoning that are already implemented in KReator. These are Bayesian logic programs originally due to Kersting et. al. [27], Markov logic networks originally due to Domingos et. al. [8], and a framework employing reasoning with maximum entropy methods [33]. Bayesian logic programs use logic programming techniques [17] in order to provide a mechanism for relational knowledge representation and employ Bayes nets [39] to perform probabilistic reasoning. Markov logic networks use classical first-order

[1] `http://www.fernuni-hagen.de/wbs/research/kreate/index.html`

logic for knowledge representation and use Markov nets [39] for reasoning. Finally, the approach on relational maximum entropy uses a relational extension of probabilistic conditional logic [43] for knowledge representation and employs the principle of maximum entropy [24] for reasoning. In the following subsections, we illustrate the use of these frameworks on a common example, the well-known burglary example [39, 4]. More examples in these different approaches will be given later in Section 5.

Example 2.1
We consider a scenario where someone—let's call him *James*—is on the road and gets a call from his neighbor saying that the alarm of James' house is ringing. James has some uncertain beliefs about the relationships between burglaries, types of neighborhoods, natural disasters, and alarms. For example, he knows that if there is a tornado warning for his home place, then the probability of a tornado triggering the alarm of his house is 0.9. He also knows that if a burglary attempt takes place, the alarm will ring with a 0.9 probability. Further he knows that if you live in bad neighborhood, then there is 0.6 probability of a burglary, whereas in an average neighborhood, there is 0.4 probability, and in a good neighborhood there is merely a 0.3 probability. A reasonable piece of information to infer from his beliefs and the given information is "What is the probability of an actual burglary?".

2.1 Bayesian Logic Programs

Bayesian logic programming is an approach to combine logic programming [17, 32] and Bayes nets [27]. Bayesian logic programs (BLPs) use a standard logic programming language and attach to each logical clause a set of probabilities, which define a conditional probability distribution of the head of the clause given specific instantiations of the body of the clause.

In contrast to first-order logic, BLPs employ an extended form of predicates and atoms. In BLPs, *Bayesian predicates* are predicates that feature an arbitrary set as possible states, i. e. not necessarily the Boolean values $\{\mathsf{true}, \mathsf{false}\}$. For example, the Bayesian predicate *bloodtype/1* may represent the blood type of a person using the possible states $S(\mathit{bloodtype}) = \{a, b, ab, 0\}$ [27]. Analogous to first-order logic, Bayesian predicates can be instantiated to *Bayesian atoms* using constants and variables and then each ground Bayesian atom represents a single random variable. If A is a Bayesian atom of the Bayesian predicate p we set $S(A) = S(p)$.

The basic structure for knowledge representation in Bayesian logic programs are *Bayesian clauses* which model probabilistic dependencies between Bayesian atoms.

Definition 2.2 (Bayesian Clause, Conditional Probability Distribution)
A *Bayesian clause* c is an expression $(H \mid B_1, \ldots, B_n)$ with Bayesian atoms $H, B_1, \ldots, B_n$. With a Bayesian clause c with the form $(H \mid B_1, \ldots, B_n)$ we associate a function $\mathsf{cpd}_c : S(H) \times S(B_1) \times \ldots \times S(B_n) \to [0, 1]$ that fulfills

$$\forall b_1 \in S(B_1), \ldots, b_n \in S(B_n) : \sum_{h \in S(H)} \mathsf{cpd}_c(h, b_1, \ldots, b_n) = 1 \quad . \tag{2.1}$$

We call cpd_c a *conditional probability distribution.* Let CPD_p denote the set of all conditional probability distributions for atoms of predicate p, i. e., it is $\mathsf{CPD}_p = \{\mathsf{cpd}_{H \mid B_1, \ldots B_n} \mid H \text{ is an atom of } p\}$.

As usual, if the body of a Bayesian clause c is empty ($n = 0$) we write c as (H) instead of $(H \mid)$ and call c a *Bayesian fact.* For a Bayesian clause $c = (H \mid B_1, \ldots, B_n)$ we abbreviate $\mathsf{head}(c) = H$ and $\mathsf{body}(c) = \{B_1, \ldots, B_n\}$. In Bayesian logic programming, variables are usually denoted with an beginning uppercase letter and constants are denoted with an initial lower-case letter. A function cpd_c for a Bayesian clause c expresses the conditional probability distribution $P(\mathsf{head}(c) \mid \mathsf{body}(c))$ and thus partially describes an underlying probability distribution P. Condition (2.1) ensures that cpd_c indeed describes a conditional probability distribution. For example, given a clause $(h(\mathsf{X}) \mid b(\mathsf{X}))$ with some Boolean predicate h condition (2.1) ensures that $P(h(\mathsf{a}) = \mathsf{true} \mid b(\mathsf{a}) = v) + P(h(\mathsf{a}) = \mathsf{false} \mid b(\mathsf{a}) = v) = 1$ for any $v \in S(b)$, some constant a, and a probability distribution P that "represents" cpd_c.

Example 2.3
We represent Example 2.1 by declaring the Bayesian predicates

$$\begin{array}{rcl}
S(\mathit{alarm/1}) & = & \{\mathsf{true}, \mathsf{false}\} \\
S(\mathit{burglary/1}) & = & \{\mathsf{true}, \mathsf{false}\} \\
S(\mathit{tornado/1}) & = & \{\mathsf{true}, \mathsf{false}\} \\
S(\mathit{lives\_in/2}) & = & \{\mathsf{true}, \mathsf{false}\} \\
S(\mathit{neighborhood/1}) & = & \{\mathsf{bad}, \mathsf{average}, \mathsf{good}\}
\end{array}$$

and defining a set $\{c_1, c_2, c_3\}$ of Bayesian clauses with:

$$\begin{array}{lcl}
c_1 & : & (\mathit{alarm}(\mathsf{X}) \mid \mathit{burglary}(\mathsf{X})) \\
c_2 & : & (\mathit{alarm}(\mathsf{X}) \mid \mathit{lives\_in}(\mathsf{X}, \mathsf{Y}), \mathit{tornado}(\mathsf{Y})) \\
c_3 & : & (\mathit{burglary}(\mathsf{X}) \mid \mathit{neighborhood}(\mathsf{X}))
\end{array}$$

For each Bayesian clause c_i, we define a function cpd_{c_i} which expresses our subjective beliefs (notice that the probabilities stated in the right column are redundant):

$$\begin{array}{rlrl}
\mathsf{cpd}_{c_1}(\mathsf{true}, \mathsf{true}) = 0.9 & & \mathsf{cpd}_{c_1}(\mathsf{false}, \mathsf{true}) = 0.1 \\
\mathsf{cpd}_{c_1}(\mathsf{true}, \mathsf{false}) = 0 & & \mathsf{cpd}_{c_1}(\mathsf{false}, \mathsf{false}) = 1 \\
\mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{true}, \mathsf{true}) = 0.9 & & \mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{true}, \mathsf{true}) = 0.1 \\
\mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{false}, \mathsf{true}) = 0 & & \mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{false}, \mathsf{true}) = 1 \\
\mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{true}, \mathsf{false}) = 0.01 & & \mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{true}, \mathsf{false}) = 0.99 \\
\mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{false}, \mathsf{false}) = 0 & & \mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{false}, \mathsf{false}) = 1 \\
\mathsf{cpd}_{c_3}(\mathsf{true}, \mathsf{bad}) = 0.6 & & \mathsf{cpd}_{c_3}(\mathsf{false}, \mathsf{bad}) = 0.4 \\
\mathsf{cpd}_{c_3}(\mathsf{true}, \mathsf{average}) = 0.4 & & \mathsf{cpd}_{c_3}(\mathsf{false}, \mathsf{average}) = 0.6 \\
\mathsf{cpd}_{c_3}(\mathsf{true}, \mathsf{good}) = 0.3 & & \mathsf{cpd}_{c_3}(\mathsf{false}, \mathsf{good}) = 0.7
\end{array}$$

For example, cpd_{c_2} expresses that our subjective belief on the probability that the alarm of a person X will go on given that we know that X lives in town Y and there is currently a tornado in Y is 0.9. Furthermore, we believe that the probability that the alarm of X will go on if we know that X lives in Y and that there is no tornado in Y is 0.01.

Considering clauses c_1 and c_2 in Example 2.3 one can see that it is possible to have multiple clauses with the same head. This means that there may be multiple causes for some effect or multiple explanations for some observation. In order to represent these kinds of scenarios the probabilities of causes or explanations have to be aggregated. BLPs use *combining rules* in order to aggregate probabilities that arise from applications of different Bayesian clauses. A combining rule cr_p for a Bayesian predicate p/n is a function $\mathsf{cr}_p : \mathfrak{P}(\mathsf{CPD}_p) \rightarrow \mathsf{CPD}_p$ that assigns to the conditional probability distributions of a set of Bayesian clauses a new conditional probability distribution that represents the *joint* probability distribution obtained from aggregating the given clauses[2]. For example, given clauses $c_1 = (b(\mathsf{X}) \mid a_1(\mathsf{X}))$ and $c_2 = (b(\mathsf{X}) \mid a_2(\mathsf{X}))$ the result $f = \mathsf{cr}_b(\{\mathsf{cpd}_{c_1}, \mathsf{cpd}_{c_2}\})$ of the combining rule cr_b is a function $f : S(b) \times S(a_1) \times S(a_2) \rightarrow [0,1]$. Appropriate choices for such functions are *average* or *noisy-or*, cf. [27].

Example 2.4
We continue Example 2.3. Suppose *noisy-or* to be the combining rule for *alarm*. Then the joint conditional probability distribution $\mathsf{cpd}_{c'}$ for

$$c' = (alarm(\mathsf{X}) \mid burglary(\mathsf{X}), lives\_in(\mathsf{X},\mathsf{Y}), tornado(\mathsf{Y}))$$

can be computed via

$$\begin{aligned}\mathsf{cpd}_{c'}(\mathsf{true}, t_1, t_2, t_3) &= 1 - (1 - \mathsf{cpd}_{c_1}(\mathsf{true}, t_1)) \cdot (1 - \mathsf{cpd}_{c_2}(\mathsf{true}, t_2, t_3)) \\ \mathsf{cpd}_{c'}(\mathsf{false}, t_1, t_2, t_3) &= 1 - \mathsf{cpd}_{c'}(\mathsf{true}, t_1, t_2, t_3)\end{aligned}$$

for any $t_1, t_2, t_3 \in \{\mathsf{true}, \mathsf{false}\}$.

A combining rule is a heuristic for estimating the probability of an event e given two causes c_1 and c_2 [39]. It has to be noted, that combining probabilities in this manner might remove the probabilistic interpretation of the resulting values as specific relationships between the causes have to be assumed. For example, using the *noisy-or* combining rule assumes *accountability* and *exception independence* of c_1 and c_2, cf. [39]. If the presumed relationships do not hold the results might be unexpected and even unwanted. Consider an extension of the BLP defined in examples 2.3 and 2.4 with the Bayesian clause

$$c_4 \quad : \quad (alarm(\mathsf{X}) \mid power\_failure(\mathsf{X})) \quad .$$

Imagine that the conditional probability distribution of c_4 assigns a probability of zero to $alarm(\mathsf{X})$ if there is a power failure. Given the evidences of both a tornado and a power failure the probability of an alarm should be determined only by considering clause c_4 and not by combining c_2 and c_4. We refer to [39] for a deeper discussion on this topic.

Now we are able to define Bayesian logic programs as follows.

Definition 2.5 (Bayesian Logic Program)
A *Bayesian logic program* B is a tuple $B = (C, D, R)$ with a (finite) set of Bayesian clauses $C = \{c_1, \ldots, c_n\}$, a set of conditional probability distributions $D = \{\mathsf{cpd}_{c_1}, \ldots, \mathsf{cpd}_{c_n}\}$ (one for each clause in C), and a set of combining rules $R = \{\mathsf{cr}_{p_1}, \ldots, \mathsf{cr}_{p_m}\}$ (one for each Bayesian predicate appearing in C) .

[2] $\mathfrak{P}(S)$ denotes the power set of a set S.

Semantics are given to Bayesian logic programs via transformation into the propositional case, i. e. into Bayes nets [39]. Given a specific (finite) universe U a Bayes net BN can be constructed by introducing a node for every grounded Bayesian atom in B. Using the conditional probability distributions of the grounded clauses and the combining rules of B a (joint) conditional probability distribution can be specified for any node in BN. If BN is acyclic this transformation uniquely determines a probability distribution P on the grounded Bayesian atoms of B which permits inference, i. e. P can be used to answer queries.

Example 2.6
Let B be the Bayesian logic program described in Example 2.3 and Example 2.4. Let $Q = (\mathit{alarm}(\mathsf{james}) = \mathsf{true} \mid E)$ with

$$\begin{aligned} E \quad = \quad \{ \quad & \mathit{lives\_in}(\mathsf{james}, \mathsf{yorkshire}) = \mathsf{true}, \mathit{tornado}(\mathsf{yorkshire}) = \mathsf{true}, \\ & \mathit{neighborhood}(\mathsf{james}) = \mathsf{average} \quad \} \end{aligned}$$

be a query to B that asks for the probability of an alarm in James' house given that he lives in an average neighborhood in Yorkshire and there is currently a tornado warning for Yorkshire. So the universe under discourse consists of the constants james and yorkshire. So, by instantiating properly and combining the conditional probability distributions of c_1 and c_2 yielding the function $\mathsf{cpd}_{c'}$ from Example 2.4 and summing over the alternatives true and false for the uncertain event $\mathit{burglary}(\mathsf{james})$, in applying c_3 we get

$$\begin{aligned} P(Q) \quad &= \quad \mathsf{cpd}_{c_3}(\mathsf{true}, \mathsf{average})\mathsf{cpd}_{c'}(\mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true}) + \\ & \quad\quad \mathsf{cpd}_{c_3}(\mathsf{false}, \mathsf{average})\mathsf{cpd}_{c'}(\mathsf{true}, \mathsf{false}, \mathsf{true}, \mathsf{true}) \\ &= \quad 0.936 \quad . \end{aligned}$$

Figure 1 illustrates the derivation of the query Q given evidence E. By omitting nodes representing clauses the derivation tree in Figure 1 can be directly transformed into a ground Bayes net that can be used to calculate the answer above.

A detailed description of the above (declarative) semantics and an equivalent procedural semantics which is based on SLD resolution are given in [27].

2.2 Markov Logic Networks

Markov logic [42] establishes a framework which combines Markov networks [39] with first-order logic to handle a broad area of statistical relational learning tasks. The Markov logic syntax complies with first-order logic[3], however each formula is quantified by an additional weight value. The semantics of a set of Markov logic formulas is determined by a probability distribution over possible worlds. A possible world ω assigns a truth value to every possible ground atom (constructible from the set of predicates and the set of constants). As we only consider finite universes we use Herbrand interpretations as possible worlds. A Herbrand interpretation is simply the set of ground atoms that are assigned the value true in this interpretation; all other ground atoms are assigned the value false. Let Ω denote the set of all possible worlds.

[3]Although Markov logic also covers functions (with some restrictions), we will omit this fact, and consider constants only.

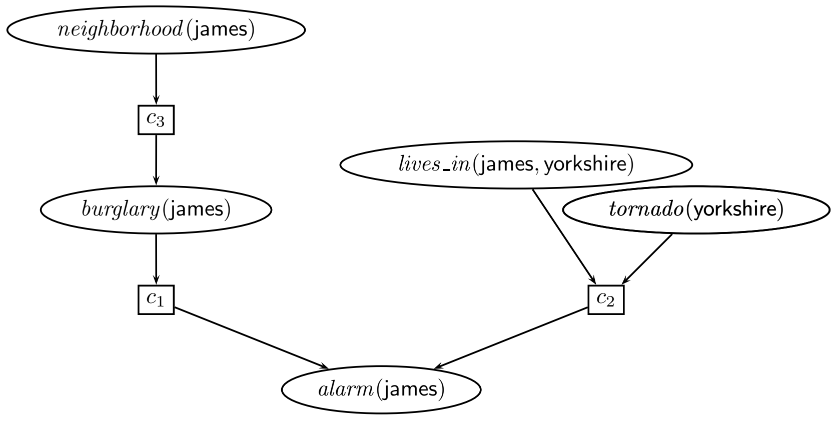


FIG. 1. The derivation for the query Q in Example 2.6.

The fundamental idea in Markov logic is that first-order formulas are not handled as hard constraints. Instead, each formula is more or less softened depending on its weight. So a possible world *may* violate a formula without necessarily receiving a zero probability. Rather a world is more probable, the less formulas it violates. A formula's weight specifies how strong the formula is, i. e. how much the formula influences the probability of a satisfying world versus a violating world. This way, the weights of all formulas influence the determination of a possible world's probability in a complex manner. One clear advantage of this approach is that Markov logic can directly handle contradictions in a knowledge base, since the (contradictory) formulas are weighted against each other anyway. Furthermore, by assigning appropriately high weight values to certain formulas, it can be enforced that these formulas will be handled as hard constraints, i. e. any world violating such a strict formula will have a zero probability. Thus, Markov logic also allows the processing of purely logical first-order formulas (leaving function symbols out of consideration).

DEFINITION 2.7 (Markov logic network)
A *Markov logic network (MLN)* L is a finite set $L = \{(F_1, w_{F_1}), \dots, (F_n, w_{F_n})\}$ of pairs (F_i, w_{F_i}) with a first-order logic formula F_i and a real value w_{F_i}, its *weight.* Together with a set of constants C it defines a *Markov network* $M_{L,C}$ as follows:

- $M_{L,C}$ contains a node for each possible grounding of each predicate appearing in L.
- $M_{L,C}$ contains an edge between two nodes (i. e. ground atoms) iff the ground atoms appear together in at least one grounding of one formula in L.
- $M_{L,C}$ contains one feature (function) for each possible grounding of each formula F_i in L. The value of the feature for a possible world ω is 1, if the ground formula is true for ω (and 0 otherwise). Each feature is weighted by the weight w_i of its respecting formula F_i.

According to the above definition, an MLN (i. e. the weighted formulas) defines a template for constructing *ground Markov networks.* For a different set C' of constants, a different ground Markov network $M_{L,C'}$ emerges from L. These ground Markov networks may vary in size, but their general structure is quite similar, e. g. the groundings of a formula F have the weight w_F in any ground Markov network of L. For each formula F in an MLN let $n_F(\omega)$ denote the number of true groundings of F in the possible world ω.

In Markov logic networks, variables are usually denoted with an initial lower-case letter and constants are denoted with a beginning uppercase letter (as opposed to the notation for Bayesian logic programming).

Example 2.8
Let $F \equiv r(\mathsf{x}) \wedge \neg s(\mathsf{x})$ be a formula over the predicates $r/1, s/1$. For a given set of constants $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, F has the groundings $F_{\mathsf{x}/\mathsf{A}} \equiv r(\mathsf{A}) \wedge \neg s(\mathsf{A})$, $F_{\mathsf{x}/\mathsf{B}} \equiv r(\mathsf{B}) \wedge \neg s(\mathsf{B})$, and $F_{\mathsf{x}/\mathsf{C}} \equiv r(\mathsf{C}) \wedge \neg s(\mathsf{C})$. So for a given possible world $\omega = \{r(\mathsf{A}), r(\mathsf{B}), r(\mathsf{C}), s(\mathsf{A})\}$, it follows $n_F(\omega) = 2$, because F has two true groundings $F_{\mathsf{x}/\mathsf{B}}, F_{\mathsf{x}/\mathsf{C}}$.

A probability distribution $P_{M_{L,C}}$ can be specified by the ground Markov network $M_{L,C}$ via the log-linear model

$$P_{M_{L,C}}(\omega) = \frac{1}{Z} \exp\left(\sum_{(F,w_F)\in L} w_F n_F(\omega)\right) \tag{2.2}$$

over possible worlds $\omega \in \Omega$ [42]. Here,

$$Z = \sum_{\omega\in\Omega} \exp\left(\sum_{(F,w_F)\in L} w_F n_F(\omega)\right)$$

is a normalization factor and the probability of an arbitrary formula A can be computed via

$$P_{M_{L,C}}(A) = \sum_{\omega\in\Omega:\omega\models A} P_{M_{L,C}}(\omega)$$

where $\omega \models A$ denotes the classical logical satisfaction relation. We illustrate the above definition with our running example.

Example 2.9
In this example, we model the relations described in Example 2.1 as an MLN (i. e. we *do not* attempt to convert the BLP from Example 2.3 to an MLN). We describe the MLN using the Alchemy syntax [31] for MLN files, therefore constant symbols must start with an upper-case letter, whereas variable symbols must have an initial lower-case letter. The "!" operator used in the predicate declarations of *lives_in* and *neighborhood* enforces that the respective variables will have mutually exclusive and exhaustive values, i. e. that every person lives in exactly one town and one neighborhood (in terms of ground atoms).

In order to appropriately represent the uncertain beliefs from Example 2.1 as a Markov logic network, the weights of formulas have to be determined. In [42] it is suggested that weights of formulas have to be learned from data. Nonetheless, in [13, 42] a heuristic is discussed that determines weights of formulas from probabilities. In [42] an interpretation of the weight w_F of a formula F is provided as the log-odd between a world where F is true and a world where F is false, other things being equal. Considering this interpretation one might choose $w_F = \log \frac{p}{1-p}$ as the weight of a formula F when p is the intended probability of F, see [13] for a discussion.

We declare the types and respective constants

$$\begin{aligned} \mathit{person} &= \{\mathsf{James}, \mathsf{Stefan}\} \\ \mathit{town} &= \{\mathsf{Freiburg}, \mathsf{Yorkshire}, \mathsf{Austin}\} \\ \mathit{hood\_status} &= \{\mathsf{Bad}, \mathsf{Average}, \mathsf{Good}\} \end{aligned}$$

the typed predicates

$$\begin{aligned} &\mathit{alarm}(\mathit{person}) \\ &\mathit{burglary}(\mathit{person}) \\ &\mathit{tornado}(\mathit{town}) \\ &\mathit{lives\_in}(\mathit{person}, \mathit{town}!) \\ &\mathit{neighborhood}(\mathit{person}, \mathit{hood\_status}!) \end{aligned}$$

and add the following weighted formulas:

$$\begin{array}{rlcl} 2.2 & \mathit{burglary}(\mathsf{x}) & \Rightarrow & \mathit{alarm}(\mathsf{x}) \\ 2.2 & \mathit{lives\_in}(\mathsf{x}, \mathsf{y}) \wedge \mathit{tornado}(\mathsf{y}) & \Rightarrow & \mathit{alarm}(\mathsf{x}) \\ 0.4 & \mathit{neighborhood}(\mathsf{x}, \mathsf{Bad}) & \Rightarrow & \mathit{burglary}(\mathsf{x}) \\ -0.4 & \mathit{neighborhood}(\mathsf{x}, \mathsf{Average}) & \Rightarrow & \mathit{burglary}(\mathsf{x}) \\ -0.8 & \mathit{neighborhood}(\mathsf{x}, \mathsf{Good}) & \Rightarrow & \mathit{burglary}(\mathsf{x}) \end{array}$$

with the heuristically determined weights $2.2 = \log \frac{0.9}{0.1}$, $0.4 = \log \frac{0.6}{0.4}$, $-0.4 = \log \frac{0.4}{0.6}$, and $-0.8 = \log \frac{0.3}{0.7}$

It has to be noted that MLNs—in contrast to BLPs (Example 2.3) and RME (Example 2.13, see below)—do not allow to express if-then-rules directly in terms of conditional probabilities [13]. So we have chosen to model the rule-like knowledge from Example 2.1 using material implications. Even though this might be a quite intuitive modelling approach at first glance, it should also be mentioned that implications are (in general) a sub-optimal choice to model conditional knowledge. As (2.2) shows, the weight of an MLN implication is effective for a probability if the implication is logically satisfied, regardless whether both the premise and the consequent hold, or just the premise fails. Therefore, a modelling approach simply using conjunctions (instead of implications) might be much more accurate in most cases.

Example 2.10

The ground formula $\mathit{burglary}(\mathsf{James}) \Rightarrow \mathit{alarm}(\mathsf{James})$ is satisfied for each world with $\{\mathit{burglary}(\mathsf{James}) = \mathsf{true} \wedge \mathit{alarm}(\mathsf{James}) = \mathit{true}\}$, as well as also for each world with $\{\mathit{burglary}(\mathsf{James}) = \mathsf{false}\}$. So the weight of this formula has exactly the same effect on the probability of a world where a burglary takes places and the alarm rings, as well as on a world where no burglary takes places. This does not model the knowledge from Example 2.1 very well, which just stated "*if* a burglary takes place, *then* the probability that the alarm rings is 0.9" and said nothing about the case that no burglary takes place. In contrast, a conditional probability $P(\mathit{alarm}(\mathsf{James}) | \mathit{burglary}(\mathsf{James})) = 0.9$ captures the knowledge from Example 2.1 adequately, because it only states a probability for $\mathit{alarm}(\mathsf{James})$ for the case that $\mathit{burglary}(\mathsf{James})$ holds.

Probabilistic inference in Markov logic is performed by calculating the conditional probability of a formula B given a formula A. That way, queries asking for the probability that a formula B is satisfied given the satisfaction of a formula A can be formulated.

Let A, B be two first-order formulas. For a given MLN L together with a set of constants C, the ground Markov network $M_{L,C}$ specifies the probability distribution $P_{M_{L,C}}$ according to (2.2), so the conditional probability of B given A is defined as:

$$P_{M_{L,C}}(B|A) = \frac{P_{M_{L,C}}(A \wedge B)}{P_{M_{L,C}}(A)} = \frac{\displaystyle\sum_{\omega\in\Omega:\omega\models A\wedge B} P_{M_{L,C}}(\omega)}{\displaystyle\sum_{\omega\in\Omega:\omega\models A} P_{M_{L,C}}(\omega)} \tag{2.3}$$

A direct calculation (2.3) is merely manageable for very small sets of constants, but intractable for domains of a more realistic size.

While the probability $P_{M_{L,C}}(B|A)$ can be approximated using Markov chain Monte-Carlo methods, the performance might still be too slow in practice [42]. Therefore, more sophisticated and efficient algorithms have been developed to perform inference in MLNs, e. g. MC-SAT and lifted belief propagation. We refer to [42] and [8] for a detailed explanation of these algorithms and other techniques regarding MLN inference.

2.3 Relational Maximum Entropy

In the following, we introduce grounding semantics for reasoning under maximum entropy in first-order probabilistic conditional logic, cf. [33]. In contrast to Bayesian logic programs or Markov logic networks which extend graph-based propositional formalisms like Bayes nets and Markov networks, this approach extends probabilistic conditional logic [37, 44] to the first-order case. In doing so, the approach discussed here is similar in spirit to the approaches undertaken in [34] and [14] which also apply Maximum Entropy reasoning on relational extensions of probabilistic conditional logic. But here we focus on *grounding techniques* for first-order probabilistic logic and particularly in resolving conflicts. The approach of relational maximum entropy (RME) relies on the central notion of a *grounding operator*. A grounding operator is a function that maps a first-order knowledge base onto a propositional one that is used for reasoning. In this paper, we give only a short overview on grounding operators and desirable properties of these.

Let $\mathcal{L}$ be a first-order language without quantifiers and without functions. We expect $\mathcal{L}$ to be typed, so let $S_{\mathcal{L}}$ be a finite set of types. Each constant c and each variable X appearing in $\mathcal{L}$ is associated with one type $\sigma \in S$ (like in Bayesian logic programming, variables are usually denoted with an beginning uppercase letter and constants are denoted with an initial lower-case letter). Furthermore, each argument of a predicate is associated with a type as well, e. g. let $p(\sigma_1, \sigma_2)$ denote a predicate which's first argument is of type σ_1 and which's second argument is of type σ_2. We assume $\mathcal{L}$ to be well-formed in the sense, that every argument of a predicate is occupied by a term (either variable or constant) of the correct type. The approach of RME relies on a first-order extension of probabilistic conditional logic, so the central structure for knowledge representation is a conditional. We allow instantiations of conditionals to be constrained by excluding unwanted combinations of constants.

Definition 2.11 (Constraint formula)
Let X, Y be some variables and $\mathsf{k}_1, \ldots, \mathsf{k}_\mathsf{l}$ some constants in $\mathcal{L}$. An *atomic constraint formula* c is either a tautology ($\top$) or an expression of the form $\mathsf{X} \neq \mathsf{Y}$ or $\mathsf{X} \notin \{\mathsf{k}_1, \ldots, \mathsf{k}_\mathsf{l}\}$. A *constraint formula* c is a finite conjunction $c = c_1 \wedge \ldots \wedge c_n$ of atomic constraint formulas $c_1, \ldots, c_n$.

Definition 2.12 (Conditional)
A *conditional* r is a structure $r = (\phi \mid \psi)[\alpha][c]$ with formulas $\phi, \psi \in \mathcal{L}$, a real value $\alpha \in [0, 1]$, and a constraint formula c. A conditional r is called *ground* iff r contains no variables. The set of all conditionals is denoted by $(\mathcal{L} \mid \mathcal{L})^{rel}$ and the set of all ground conditionals is denoted by $(\mathcal{L} \mid \mathcal{L})^{rel}_U$.

A conditional $(\phi \mid \psi)[\alpha][c]$ describes some kind of default knowledge like "When ψ then (normally) ϕ". While α represents the intended probability of the conditional to be true, c represents constraints to be respected when grounding the conditional. Formal semantics to conditionals will be given below. If the premise ψ of a conditional is tautological ($\psi \equiv \top$) we write $(\phi)[\alpha][c]$ instead of $(\phi \mid \psi)[\alpha][c]$. When c is tautological we write $(\phi \mid \psi)[\alpha]$ instead of $(\phi \mid \psi)[\alpha][c]$. A set KB of conditionals is called a *knowledge base.*

Example 2.13
We represent Example 2.1 in the RME approach as a knowledge base KB. Let the set S of types be given by $S = \{Person, Town, HoodStatus\}$ which, respectively, refer to persons, towns, and status of neighborhood. Then we define following constants of the respective type:

$$\begin{aligned} Person &= \{\mathsf{james}, \mathsf{stefan}\} \\ Town &= \{\mathsf{freiburg}, \mathsf{yorkshire}, \mathsf{austin}\} \\ HoodStatus &= \{\mathsf{bad}, \mathsf{average}, \mathsf{good}\} \end{aligned}$$

The set P contains the following predicates:

$$\begin{array}{l} alarm(Person) \\ burglary(Person) \\ tornado(Town) \\ lives\_in(Person, Town) \\ neighborhood(Person, HoodStatus) \end{array}$$

Then our knowledge base KB can be defined by $KB = \{c_1, \ldots, c_7\}$ with

$$\begin{array}{lll} c_1 & : & (alarm(\mathsf{X}) \mid burglary(\mathsf{X}))\ [0.9] \\ c_2 & : & (alarm(\mathsf{X}) \mid lives\_in(\mathsf{X}, \mathsf{Y}), tornado(\mathsf{Y}))\ [0.9]\ \} \\ c_3 & : & (burglary(\mathsf{X}) \mid neighborhood(\mathsf{X}, \mathsf{bad}))\ [0.6] \\ c_4 & : & (burglary(\mathsf{X}) \mid neighborhood(\mathsf{X}, \mathsf{average}))\ [0.4] \\ c_5 & : & (burglary(\mathsf{X}) \mid neighborhood(\mathsf{X}, \mathsf{good}))\ [0.3] \\ c_6 & : & (neighborhood(\mathsf{X}, \mathsf{Z}) \mid neighborhood(\mathsf{X}, \mathsf{Y}))\ [0.0]\ [\mathsf{Y} \neq \mathsf{Z}] \\ c_7 & : & (lives\_in(\mathsf{X}, \mathsf{Z}) \mid lives\_in(\mathsf{X}, \mathsf{Y}))\ [0.0]\ [\mathsf{Y} \neq \mathsf{Z}] \end{array}$$

Notice, that the constraint formulas of conditionals c_6 and c_7 ensure that only one value of the second argument of *neighborhood* resp. *lives_in* is valid at any time.

Semantics are given to a knowledge base KB by grounding KB with a *grounding operator* (GOP).

Definition 2.14 (Grounding operator)
A *grounding operator* (GOP) $\mathcal{G}$ is a function $\mathcal{G} : \mathfrak{P}((\mathcal{L} \mid \mathcal{L})^{rel}) \to \mathfrak{P}((\mathcal{L} \mid \mathcal{L})^{rel}_U)$ which maps knowledge bases to ground knowledge bases.

Note, that a ground knowledge base is equivalent to a purely propositional knowledge base by treating ground atoms as propositions. Here, we do not go into details in strategies for grounding a knowledge base but suppose $\mathcal{G}$ to be some reasonable grounding operator as given. In [33] the *naive*-, *cautious*-, *conservative*-, and *specificity*-grounding strategies are presented and analyzed and a more elaborate treatment of grounding strategies for RME is given. For now, we just consider the following example (adapted from [6]) of a grounding operator. We will also revisit this example in Section 5.2 for an in-depth comparison of different modeling approaches.

Example 2.15
Let KB be given by $KB = \{c_1, c_2, c_3\}$ with

$$\begin{aligned} c_1 &: (\mathit{likes}(\mathsf{X}, \mathsf{Y}))[0.9] \\ c_2 &: (\mathit{likes}(\mathsf{X}, \mathsf{fred}))[0.3] \\ c_3 &: (\mathit{likes}(\mathsf{clyde}, \mathsf{fred}))[1.0] \end{aligned}$$

The knowledge base contains a general statement (c_1) that represents the probability of an elephant liking its keeper, and two more specific statements (c_2 resp. c_3) that model the relationships for some exceptional individuals Clyde and Fred. Grounding KB using universal instantiation $\mathcal{G}_{univ}$, i. e. grounding each conditional with any possible combination of constants, yields an inconsistent ground knowledge base $\mathcal{G}_{univ}(KB)$ as $\mathcal{G}_{univ}(KB)$ contains, for instance, both $(\mathit{likes}(\mathsf{clyde}, \mathsf{fred}))[0.3]$ and $(\mathit{likes}(\mathsf{clyde}, \mathsf{fred}))[1]$. Nonetheless, KB makes perfect sense from a commonsensical point of view as, for instance, rule c_2 should be treated as an exception to c_1 and inhibiting the instantiation of Y with the constant fred in c_1. One approach to avoid these inconsistencies is to completely ignore individuals that already appear within the knowledge base when instantiating the conditionals. This *cautious grounding* operator $\mathcal{G}_{ca}$ is formalized in [33]. Consider that language contains the elephants Clyde, Dumbo and Tuffi and the keepers Fred and Hank. Then the resulting ground knowledge base $\mathcal{G}_{ca}(KB)$ of KB can be given as

$$\begin{aligned} \mathcal{G}_{ca}(KB) \quad = \quad \{ \quad & (\mathit{likes}(\mathsf{dumbo}, \mathsf{hank}))[0.9], \\ & (\mathit{likes}(\mathsf{tuffi}, \mathsf{hank}))[0.9], \\ & (\mathit{likes}(\mathsf{dumbo}, \mathsf{fred}))[0.3], \\ & (\mathit{likes}(\mathsf{tuffi}, \mathsf{fred}))[0.3], \\ & (\mathit{likes}(\mathsf{clyde}, \mathsf{fred}))[1.0] \quad \}. \end{aligned}$$

Observe, for instance, that $\mathcal{G}_{ca}$ does not instantiate c_1 using the constant *clyde*, because *clyde* already appears in the knowledge base.

So let $\mathcal{G}$ be some well-defined grounding operator and KB be a knowledge base. Then $\mathcal{G}(KB)$ is ground and can be treated as a propositional knowledge base. As before, a Herbrand interpretation ω of our relational language $\mathcal{L}$ is any set of ground atoms of $\mathcal{L}$ and let Ω be the set of all Herbrand interpretations for $\mathcal{L}$. Semantics are given to a knowledge base KB with means of probability distributions $P : \Omega \to [0, 1]$ which map interpretations to probabilities. A probability distribution P can be extended to ground formulas $A \in \mathcal{L}$ by defining (also as before)

$$P(A) = \sum_{\omega\in\Omega,\omega\models A} P(\omega)$$

where $\omega \models A$ denotes the classical logical satisfaction relation. Then, given a knowledge base KB and a grounding operator $\mathcal{G}$, we say that a probability distribution P satisfies KB under $\mathcal{G}$, denoted by $P \models_{\mathcal{G}} KB$, if and only if

$$\forall(\phi^* \mid \psi^*)[\alpha] \in \mathcal{G}(KB) : P(\phi^* \mid \psi^*) = \alpha \quad .$$

This means, that $P \models_{\mathcal{G}} KB$ holds if the conditional probability $P(\phi^* \mid \psi^*)$ equals α for any ground conditional $(\phi^* \mid \psi^*)[\alpha]$ in $\mathcal{G}(KB)$. To perform reasoning we employ the principle of maximum entropy [24, 20, 43, 44]. The *entropy* H of a probability distribution P is defined as

$$H(P) = -\sum_{\omega\in\Omega} P(\omega)\log P(\omega) \quad .$$

By selecting the one probability distribution P that yields maximum entropy, i. e., by computing

$$P^{ME}_{\mathcal{G},KB} = \arg\max_{P\models_{\mathcal{G}}KB} H(P) \tag{2.4}$$

we obtain the most unbiased representation of the knowledge in $\mathcal{G}(KB)$, cf. [24, 20, 43, 44] for the formal properties and uniqueness theorems. The approach of maximum entropy yields a model-based inference procedure and reasoning on KB is now solely performed using the probability distribution $P^{ME}_{\mathcal{G},KB}$. Then a conditional $Q \in (\mathcal{L} \mid \mathcal{L})^{rel}$ is *fulfilled under the grounding* $\mathcal{G}(R)$ in the knowledge base KB, denoted by $KB \models^{ME}_{\mathcal{G}} Q$, if it holds

$$\begin{aligned} KB \models^{ME}_{\mathcal{G}} Q \quad &\text{iff} \quad P^{ME}_{\mathcal{G},KB} \models \mathcal{G}(\{Q\}) \\ &\text{iff} \quad \forall(\phi^* \mid \psi^*)[\alpha] \in \mathcal{G}(\{Q\}) : P^{ME}_{\mathcal{G},KB}(\phi^* \mid \psi^*) = \alpha \quad . \end{aligned}$$

This means, that $KB \models^{ME}_{\mathcal{G}} Q$ holds if the conditional probability of every ground instance $(\phi^* \mid \psi^*)$ of Q wrt. $\mathcal{G}$ equals α in $P^{ME}_{\mathcal{G},KB}$. Figure 2 gives an overview on the RME approach introduced above.

EXAMPLE 2.16
Continuing Example 2.13 consider the query $Q = (alarm(\mathsf{james}) \mid E)$ with

$$E = \{lives\_in(\mathsf{james}, \mathsf{austin}), tornado(\mathsf{austin}), neighborhood(\mathsf{james}, \mathsf{average})\} \quad .$$

As before, Q asks for the probability of $alarm(\mathsf{james})$ being true given that James lives in Austin, there is currently a tornado warning for Austin and the neighborhood of

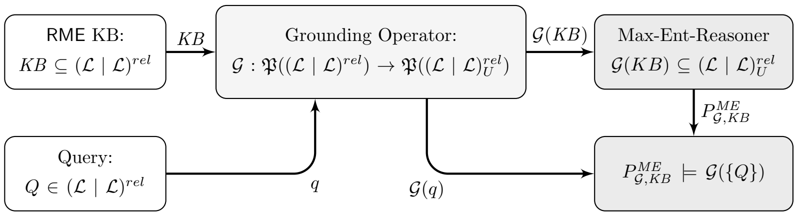

RME KB:
$KB \subseteq (\mathcal{L} \mid \mathcal{L})^{rel}$
KB
Grounding Operator:
$\mathcal{G} : \mathfrak{P}((\mathcal{L} \mid \mathcal{L})^{rel}) \rightarrow \mathfrak{P}((\mathcal{L} \mid \mathcal{L})^{rel}_U)$
$\mathcal{G}(KB)$
Max-Ent-Reasoner
$\mathcal{G}(KB) \subseteq (\mathcal{L} \mid \mathcal{L})^{rel}_U$
$P^{ME}_{\mathcal{G},KB}$
Query:
$Q \in (\mathcal{L} \mid \mathcal{L})^{rel}$
q
$\mathcal{G}(q)$
$P^{ME}_{\mathcal{G},KB} \models \mathcal{G}(\{Q\})$

FIG. 2. Overview on reasoning with RME.

James can be considered as an average neighborhood. As Q is already ground there is no need to compute $\mathcal{G}(\{Q\})$. So we go on by grounding KB using some chosen grounding operator. In this example, the result of this grounding is independent of the actual chosen grounding operator (as there is no conflicting information on a syntactical level), hence we can use universal instantiation. The following is a short excerpt of the set of formulas $\mathcal{G}(KB)$ yields to:

$$\begin{aligned}
c_{1,1} &: (\mathit{alarm}(\mathsf{james}) \mid \mathit{burglary}(\mathsf{james}))\ [0.9]\\
c_{1,2} &: (\mathit{alarm}(\mathsf{stefan}) \mid \mathit{burglary}(\mathsf{stefan}))\ [0.9]\\
c_{2,1} &: (\mathit{alarm}(\mathsf{james}) \mid \mathit{lives\_in}(\mathsf{james}, \mathsf{freiburg}), \mathit{tornado}(\mathsf{freiburg}))\ [0.9]\\
c_{2,2} &: (\mathit{alarm}(\mathsf{james}) \mid \mathit{lives\_in}(\mathsf{james}, \mathsf{austin}), \mathit{tornado}(\mathsf{austin}))\ [0.9]\\
&\ldots
\end{aligned}$$

For $\mathcal{G}(KB)$ we now compute $P^{ME}_{\mathcal{G},KB}$ by selecting the one probability distribution with maximum entropy that satisfies $\mathcal{G}(KB)$, cf. Equation (2.4). Remember that $\mathcal{G}(KB)$ is ground and as such can be treated as propositional knowledge base, so computing $P^{ME}_{\mathcal{G},KB}$ can be done using traditional methods for computing the one model with maximum entropy for propositional models, see e. g. [44]. Then the conditional probability of $\mathit{alarm}(\mathsf{james})$ given E can be directly calculated to

$$P^{ME}_{\mathcal{G},KB}(\mathit{alarm}(\mathsf{james}) \mid E) = 0.8951 \quad .$$

3 KReator: Overview and System Architecture

So far, we described three different approaches for combining first-order representations of knowledge with probabilistic reasoning. These approaches only serve as representatives for a huge variety of different proposals to statistical relational learning and inductive logic programming. But still, one can note that even these three approaches differ significantly in both their representation and reasoning behavior; even in such unimportant manners such as naming conventions for variables and constants (we deliberately used the common notation for the presented approaches to highlight this problem). Consequently, as a researcher the problem of evaluating and comparing these formalisms becomes a tedious task, especially when implementations are prototypical and hard to use. In this section, we give an overview on the KReator system that aims at alleviating common tasks when working with formalisms for statistical relational learning or inductive logic programming.

3.1 Overview

KReator[4] is an integrated development environment for representing, reasoning, and learning with relational probabilistic knowledge. Still being in development KReator aims to become a versatile toolbox for researchers and knowledge engineers in the field of statistical relational learning. Coming with an intuitive graphical user interface KReator's core functionalities allow the knowledge engineer to specify knowledge bases, ask queries to knowledge bases, and to learn knowledge bases from data. At this time, KReator supports these tasks using Bayesian logic programs, Markov logic networks, and relational maximum entropy (cf. Sections 2.1, 2.2, and 2.3). The main advantage of using KReator (instead of using prototypical implementations of these formalisms itself) is the common and unified way to address such tasks as described above. Furthermore, KReator features diverse user friendly amenities like project management, scripting, syntax highlighting, or LaTeX output, which ease the work on knowledge representation, reasoning, and learning. In the rest of this and the subsequent section we will discuss several of KReator's features in more detail.

3.2 Design and Architecture

KReator is written in Java [19] and designed using the object-oriented programming paradigm. By using Java as programming language KReator is platform independent and runs on any system with a Java runtime environment. It facilitates several architectural and design patterns [16] such as model-view control, abstract factories, and command patterns. Central aspects of the design of KReator are *modularity*, *extensibility*, *usability*, *reproducibility*, and its intended *application in scientific research*.

Modularity and Extensibility KReator is modular and extensible with respect to several components. In the following we discuss just two important aspects. First, KReator's internal logic is strictly separated from the user interface by an abstract command layer. This separation not only guarantees a clean and extendable program structure, but it also allows to make KReator's core functionalities available to the user by two different user interfaces: an intuitive-to-use graphical user interface, as well as a powerful command-line interface (the KReator *console*) which processes commands in JavaScript syntax (see Section 4.1). Second, KReator was designed to support many different approaches for relational knowledge representation, cf. Section 2. As a consequence, KReator features very abstract notions of concepts like knowledge bases, queries and data sets that can be implemented by a specific approach. Each implemented knowledge representation approach is encapsulated in an individual plug-in (so at the moment, there are BLP, MLN, and RME plug-ins). This way, integrating a certain approach into KReator is most flexible and further approaches will be added in the near future. As KReator is open-source and available under the GNU General Public License researchers are also encouraged to implement their approaches themselves.

[4] The "KR" in KReator stands for "Knowledge Representation" and the name KReator indicates its intended usage as a development environment for knowledge engineers.

Usability An important design aspect of KReator and especially of the graphical user interface is usability. While prototypical implementations of specific approaches to relational probabilistic knowledge representation (and approaches for any problem in general) are essential for validating results and evaluation, these software solutions are often very hard to handle and differ significantly in their usage. Especially when one wants to compare different solutions these tools do not offer an easy access for new users. KReator features a common and simple interface to different approaches of relational probabilistic knowledge representation within a single application. The work with KReator is furthermore eased by several amenities known from integrated development environments for programming languages like syntax highlighting and syntax correction.

Reproducibility KReator records every user operation (no matter whether it was caused by GUI interaction or by a console input) and its result in a *report*. Since all operations are reported in JavaScript syntax (see Section 4.1), the report itself represents a valid sequence of JavaScript commands. Therefore the whole report or parts of it can be saved as a JavaScript file, which can be executed anytime to repeat the recorded operations. So the KReator report allows to retrace and reproduce all the steps taken when experimenting with knowledge bases. That way, KReator supports the user to document the results of experiments and the actions which led to these results in an exact manner.

Application in Scientific Research Both usability and reproducibility are important aspects when designing a tool for conducting scientific research. Besides that, other important features are also provided within KReator. For example, KReator can export knowledge base files as formatted LaTeX output, making the seamless processing of example knowledge bases in scientific publications very convenient.

3.3 Used frameworks

KReator makes use of well-established software frameworks to process some of the supported knowledge representation formalisms. Performing inference on MLNs is handled entirely by the Alchemy[5] software package [31], a console-based tool for processing Markov logic networks. In order to make MLN inference with Alchemy transparent to use, KReator has to accomplish several steps in the background: First, all user input data (i. e. the query and the accompanied evidence) is converted into the appropriate Alchemy syntax (see Section 4.2 for more details) and placed in temporary files so that Alchemy can work with it. Next, KReator calls the Alchemy binary as an external application with the appropriate command-line parameters which control several important aspects of the inference process (e. g. what inference algorithm to use). While Alchemy calculates the answer to the query, KReator monitors its execution for any potential problems. After Alchemy has successfully finished its calculations, KReator reads in the Alchemy output file, parses its content, and presents the result in a user-friendly format. That way, Alchemy is seamlessly integrated into KReator and the user does not have to deal with Alchemy software directly.

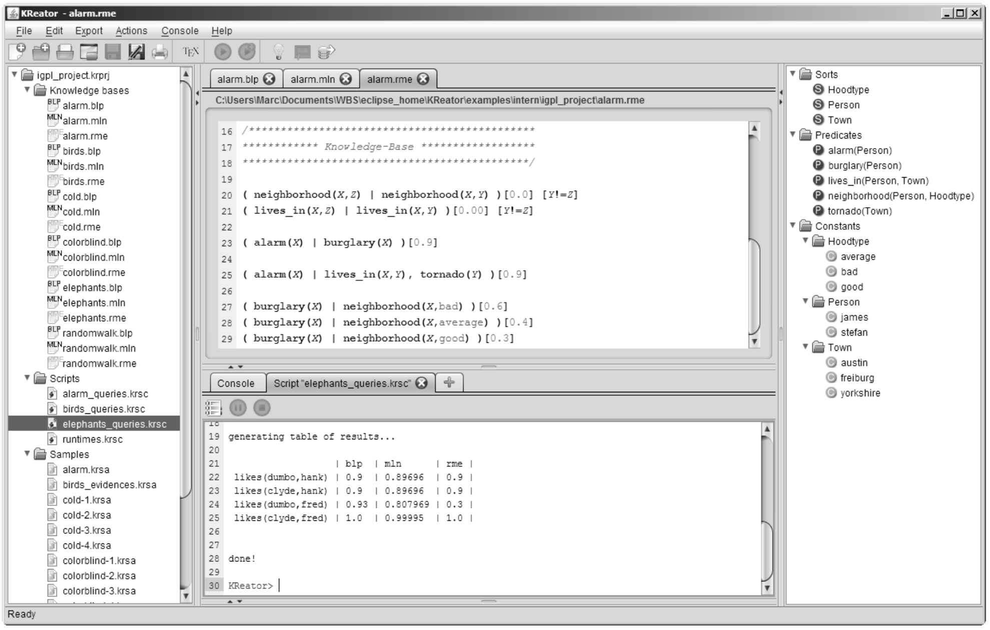
To process ground RME knowledge bases, an appropriate reasoner for maximum entropy must be utilized. KReator does not directly interact with a certain reasoner.

[5]http://alchemy.cs.washington.edu/

Instead, KReator uses a so-called ME-adapter to communicate with a (quite arbitrary) MaxEnt-reasoner. That is, to connect a certain MaxEnt-reasoner to KReator, an appropriate implementation of the ME-adapter interface must be supplied that handles the direct communication with the reasoner's API. Currently, such an adapter is supplied for the SPIRIT reasoner [44]. SPIRIT[6] is a tool for processing (propositional) conditional probabilistic knowledge bases using maximum entropy methods. An appropriate adapter for the MEcore reasoner [11] has also been developed (in terms of a proof-of-concept).

4 Usage

KReator comes with a graphical user interface and an integrated console-based interface. The main view of KReator (see Figure 3) is divided into the menu and toolbars, as well as four main panels: the project panel, the editor panel, the outline panel, the console panel. Nearly every property of the graphical user interface can



```
/*********************************************
************ Knowledge-Base *****************
*********************************************/

( neighborhood(X,Z) | neighborhood(X,Y) )[0.0]  [Y!=Z]
( lives_in(X,Z) | lives_in(X,Y) )[0.00]  [Y!=Z]

( alarm(X) | burglary(X) )[0.9]

( alarm(X) | lives_in(X,Y), tornado(Y) )[0.9]

( burglary(X) | neighborhood(X,bad) )[0.6]
( burglary(X) | neighborhood(X,average) )[0.4]
( burglary(X) | neighborhood(X,good) )[0.3]
```

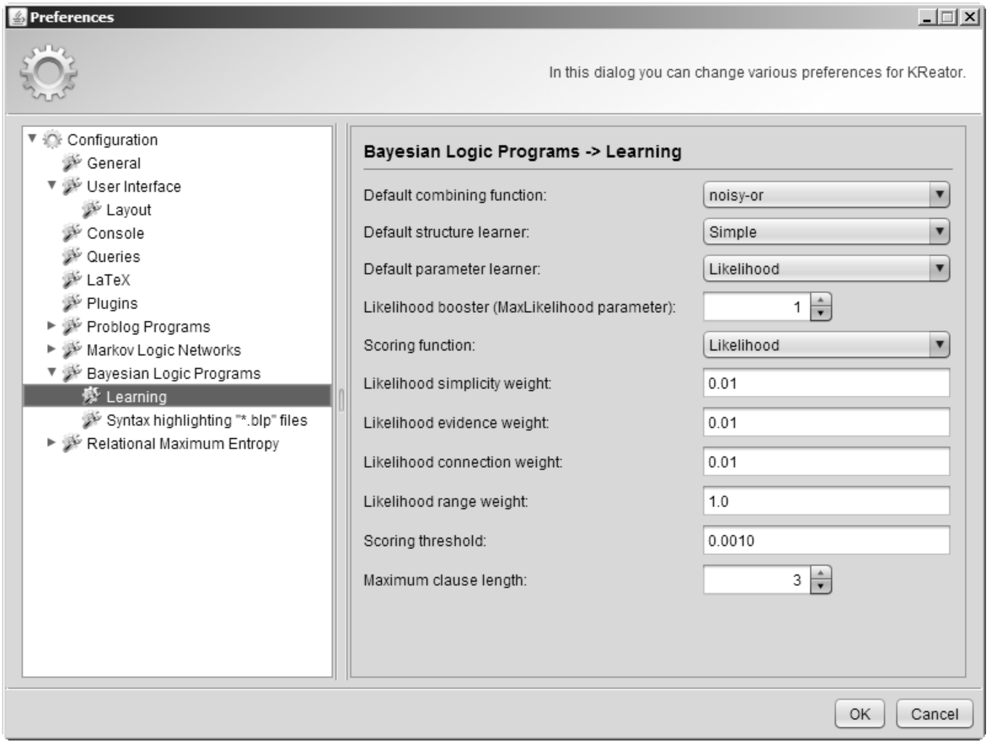
```
generating table of results...

                   | blp  | mln      | rme |
 likes(dumbo,hank) | 0.9  | 0.89696  | 0.9 |
 likes(clyde,hank) | 0.9  | 0.89696  | 0.9 |
 likes(dumbo,fred) | 0.93 | 0.807969 | 0.3 |
 likes(clyde,fred) | 1.0  | 0.99995  | 1.0 |


done!

KReator> 
```

Fig. 3. KReator – Main window

be conveniently configured using KReator's preferences dialog (Figure 4). This dialog also provides access to the configuration of special features of each knowledge representation formalism. Furthermore, every configuration property can also be modified using JavaScript so that different configurations can be tested easily in script files.

The project panel KReator structures its data into projects which may contain

[6] http://www.fernuni-hagen.de/BWLOR/spirit_int/


Preferences

In this dialog you can change various preferences for KReator.

- Configuration
  - General
  - User Interface
    - Layout
  - Console
  - Queries
  - LaTeX
  - Plugins
  - Problog Programs
  - Markov Logic Networks
  - Bayesian Logic Programs
    - Learning
    - Syntax highlighting "*.blp" files
  - Relational Maximum Entropy

Bayesian Logic Programs -> Learning

Default combining function:	noisy-or
Default structure learner:	Simple
Default parameter learner:	Likelihood
Likelihood booster (MaxLikelihood parameter):	1
Scoring function:	Likelihood
Likelihood simplicity weight:	0.01
Likelihood evidence weight:	0.01
Likelihood connection weight:	0.01
Likelihood range weight:	1.0
Scoring threshold:	0.0010
Maximum clause length:	3

Fig. 4. KReator – Preferences dialog

knowledge bases, scripts written in JavaScript (see below), query collections for knowledge bases, and evidence or sample files. Although all types of files can be opened independently in KReator, projects can help the knowledge engineer to organize his work. The project panel of KReator (seen in the left in Figure 3) gives a complete overview on the project the user is currently working on.

The editor panel All files supported by KReator can be viewed and edited in the editor panel (seen in the upper middle in Figure 3). Multiple files can be opened at the same time and the editor supports editing knowledge bases and the like with syntax highlighting, syntax check, and other features normally known from development environments for programming languages, e. g. word completion. KReator's syntax highlighting feature does not simply work on a fixed set of keywords of the respective knowledge bases syntax, but it is based on an in-depth analysis of the knowledge base file. This allows to precisely identify all structural elements of a knowledge base and highlight them appropriately in the editor, so the user can directly distinguish between predicates, constants, variables, etc. Furthermore, syntactical errors in a knowledge base file can be detected while the file is edited and the affected part becomes highlighted immediately, providing an instantaneous feedback to the user.

The outline panel The outline panel (seen in the right in Figure 3) gives an structured overview on the currently viewed file in the editor panel. To provide such a structural view of a text file, the file's content has to be analyzed and interpreted on-the-fly and internally represented by appropriate KReator data-structures. Performing a mouse-click on an element in the outline highlights all occurrences of this element in the file and successively navigates to the appropriate position. If the file is a knowledge base, the outline shows all available information on the logical components of the knowledge base. The tree-like structure presents types (if the knowledge base uses a typed language), predicates (and, in case of BLPs, their states), and constants (subdivided by types, if applicable) in a clearly arranged fashion. In case of an evidence or sample file, the outline provides two different views: On the one hand, the outline shows a sorted list of the predicates and constants which compose the ground literals of the sample. On the other hand, the outline provides a view of the ground literals, which can be sorted either by predicate (i. e. grouping all literals of a certain predicate) or by constant (i. e. grouping all literals that contain a certain constant).

The console panel The console panel (seen at the lower middle in Figure 3) contains KReator's command-line interface. The console offers access to KReator's functionalities just using textual commands, e. g. to query a knowledge bases. As a matter of fact, the console is a live interpreter for JavaScript (see below).

The report panel The console panel can be switched to the report panel, in which every action executed in KReator (regardless of caused by GUI interaction or console input) is recorded as a JavaScript command, along with its output (formatted as JavaScript comment). The whole report or parts of it can easily be saved as script file and executed again when experiments have to be repeated and results have to be reproduced.

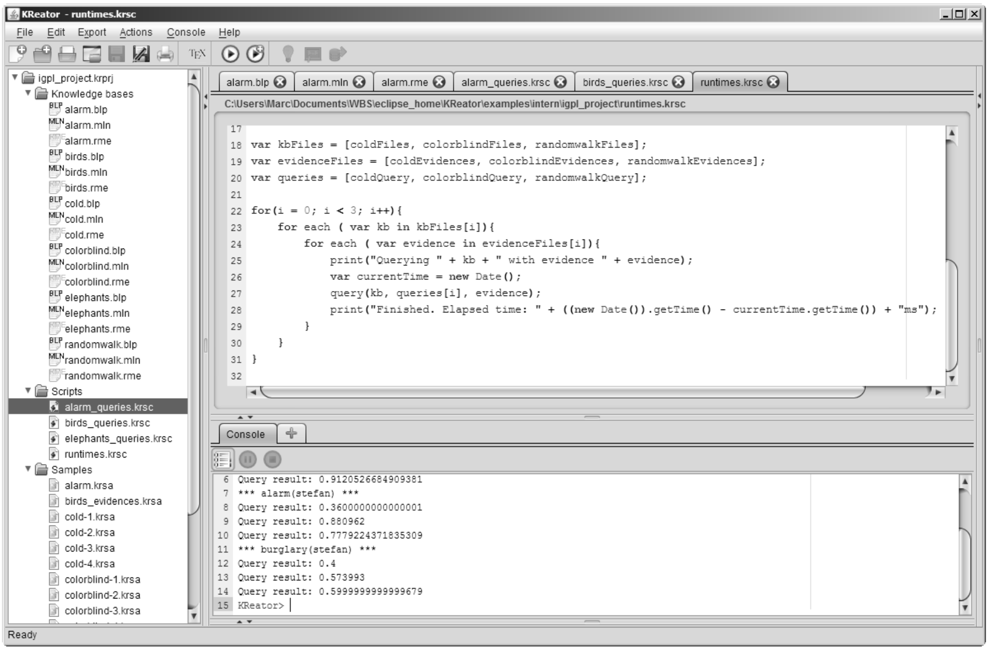
4.1 Employment of JavaScript

KReator scripting facilities are completely based on the Mozilla Rhino JavaScript engine[7] (replacing the proprietary KReatorScript language used in previous versions of KReator). KReator introduces appropriate JavaScript commands for all its high-level functionalities (see Figure 5 for some JavaScript lines). Therefore, every sequence of working steps can be expressed as an appropriate command sequence in a JavaScript file. Thus, the (re-)utilization of scripts can clearly increase the efficiency and productivity when working with KReator. As mentioned above, the input-console and report work hand in hand with KReator's scripting functionality, making the employment of JavaScript a strong instrument in the whole working process. Besides that, the utilization of scripts plays an important role in the active development of the KReator software, since it allows efficient and sustainable testing of high-level functionalities during the whole development process.

4.2 Querying a Knowledge Base

One of the most important tasks when working with knowledge bases is to address queries to a knowledge base, i. e. to infer knowledge. For that reason, KReator

[7] `http://www.mozilla.org/rhino/`



C:\Users\Marc\Documents\WBS\eclipse_home\KReator\examples\intern\igpl_project\runtimes.krsc

```

var kbFiles = [coldFiles, colorblindFiles, randomwalkFiles];
var evidenceFiles = [coldEvidences, colorblindEvidences, randomwalkEvidences];
var queries = [coldQuery, colorblindQuery, randomwalkQuery];

for(i = 0; i < 3; i++){
    for each ( var kb in kbFiles[i]){
        for each ( var evidence in evidenceFiles[i]){
            print("Querying " + kb + " with evidence " + evidence);
            var currentTime = new Date();
            query(kb, queries[i], evidence);
            print("Finished. Elapsed time: " + ((new Date()).getTime() - currentTime.getTime()) + "ms");
        }
    }
}

```

```
Query result: 0.9120526684909381
*** alarm(stefan) ***
Query result: 0.3600000000000001
Query result: 0.880962
Query result: 0.7779224371835309
*** burglary(stefan) ***
Query result: 0.4
Query result: 0.573993
Query result: 0.5999999999999679
KReator> |
```

Fig. 5. JavaScript file and console output

provides several functionalities which simplify the dealing with queries and make it more efficient.

KReator permits the processing of queries expressed in a *unified query syntax*. This query syntax abstracts from the respective syntax which is necessary to address a "native" query to a BLP, MLN, or RME knowledge base (and which also depends on the respective inference engine). That way, a query in unified syntax can be passed to an appropriate BLP, MLN, and RME knowledge base as well. The idea behind this functionality is, that some knowledge (cf. Example 2.1) can be modeled in different knowledge representation approaches (cf. Example 2.3, Example 2.9, and Example 2.13) and the user is able to compare these approaches in a more direct way. Such a comparison can then be done by formulating appropriate queries in unified syntax, passing them to the different knowledge bases, and finally analyzing the different answers, i. e. the probabilities.

A KReator query in unified syntax consists of two parts: In the "head" of the query there are one or more ground atoms whose probabilities shall be determined. The "body" of the query is composed of several evidence atoms. For each supported knowledge representation formalism, KReator must convert a query in unified syntax in the exact syntax required by the respective inference engine. Among other things, this includes e. g. the conversion from lower case constants to upper case ones (and variables, vice versa), as required by the Alchemy tool for processing MLNs. KReator also converts the respective output results to present them in a standardized format to the user. Figure 6 illustrates the processing of a query in unified syntax.

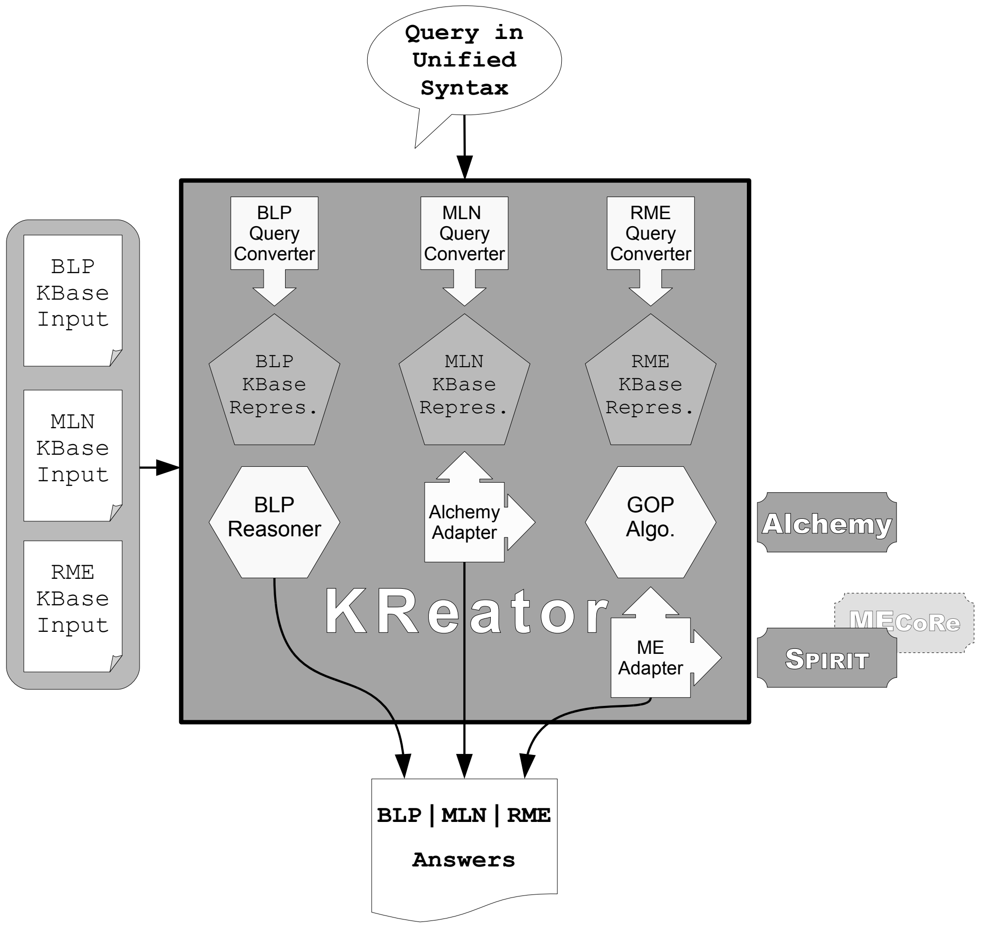
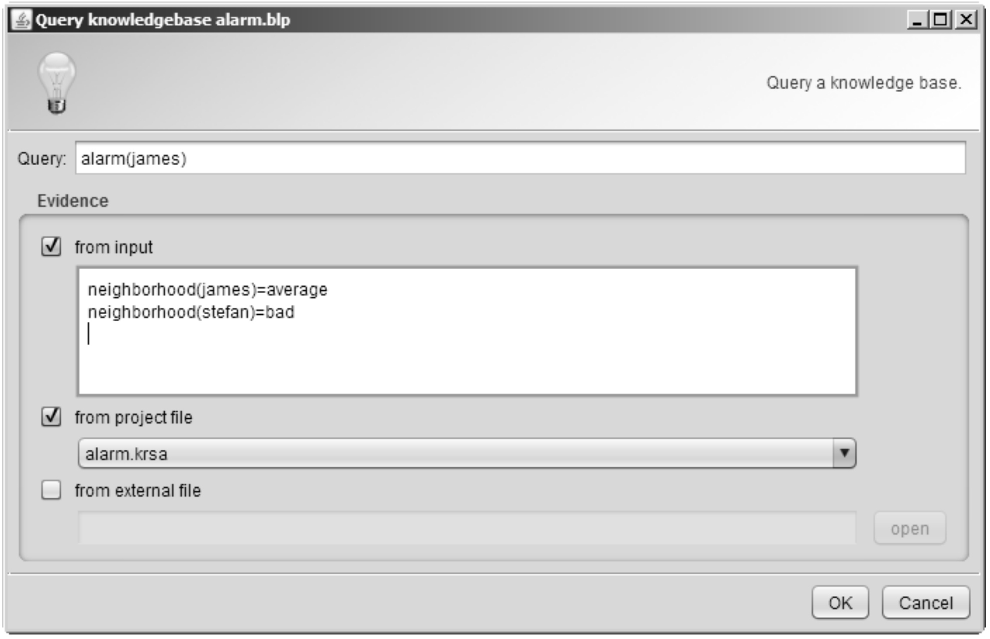


Fig. 6. Processing query in unified syntax

KReator offers the user an easy way to address a query to a knowledge base, simply by calling its query dialogue. In this dialogue (Figure 7), the user can input the atoms to be queried and he can conveniently specify the evidence. Since evidence usually consists of several atoms and is often reused for different queries, the user has the option to specify a file which contains the evidence to be to considered.

The unified query syntax constitutes a compromise between the querying capabilities of the different knowledge representation formalisms. Therefore, some individual features of each formalism cannot be expressed in unified syntax. For this reason, KReator additionally offers a direct access to the querying capabilities of each inference engine. This is realized by corresponding console commands which take a query in "native" syntax as an argument. Besides the capability of passing individual (i. e. ad-hoc) queries to a knowledge base, KReator also supports so-called *query collections*. A query collections is a file which contains several queries (either in uni-


Query knowledgebase alarm.blp

Query a knowledge base.

Query: alarm(james)

Evidence

[✓] from input

```
neighborhood(james)=average
neighborhood(stefan)=bad
```

[✓] from project file

alarm.krsa

[ ] from external file

FIG. 7. Querying a knowledge base

fied or native syntax). Such a query collection can be passed to a knowledge base, so that all included queries are processed one after another. That way, KREATOR supports a persistent handling and batch-style processing of queries.

EXAMPLE 4.1
Continuing our previously introduced burglary example (cf. the knowledge bases from Example 2.3, Example 2.9, and Example 2.13) consider the following evidence:

$$\textit{lives\_in}(\mathsf{james}, \mathsf{yorkshire}), \textit{lives\_in}(\mathsf{stefan}, \mathsf{freiburg}), \textit{burglary}(\mathsf{james}),$$
$$\textit{tornado}(\mathsf{freiburg}), \textit{neighborhood}(\mathsf{james}, \mathsf{average}), \textit{neighborhood}(\mathsf{stefan}, \mathsf{bad})$$

Notice, that we use a slightly different variant for the BLP as represented in Example 2.3 with a binary version of the *neighborhood* predicate. Table 1 shows three queries and their respective probabilities inferred from each of the example knowledge bases[8]. Each of the three knowledge bases represents Example 2.1 by a different knowledge representation approach. Nevertheless, the inferred probabilities are quite similar, except for some differences for the third query.

4.3 Learning a Knowledge Base from Data

Besides querying (see previous subsection) another important aspect for statistical relational learning is quite obviously the task of *learning* which describes the process

[8]All MLN results in this paper have been calculated using the default Alchemy settings: MC-SAT inference algorithm with a maximum of 1000 MCMC sampling steps.

	BLP	MLN	RME
$alarm$(james)	0.900	0.971	0.913
$alarm$(stefan)	0.954	0.967	0.922
$burglary$(stefan)	0.400	0.570	0.650

TABLE 1. Exemplary queries on different representations of the burglary example.

of building up a knowledge base from a set of data sets. For example, consider the following set of ground facts:

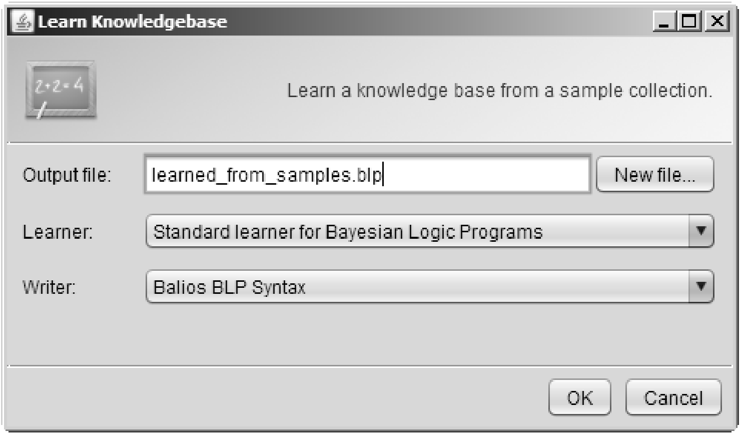
$$\begin{array}{l} alarm(\mathsf{anna}), alarm(\mathsf{carl}), lives\_in(\mathsf{carl}, \mathsf{toronto}), burglary(\mathsf{carl}) \\ tornado(\mathsf{austin}), neighborhood(\mathsf{anna}, \mathsf{average}), neighborhood(\mathsf{bob}, \mathsf{bad}) \\ lives\_in(\mathsf{anna}, \mathsf{austin}), lives\_in(\mathsf{bob}, \mathsf{hongkong}), burglary(\mathsf{anna}), \ldots \end{array}$$

The aim of learning is to discover relationships between the individual pieces of information in such a data set and generalize them in such a manner that a knowledge base of the form as in Example 2.3, Example 2.9, or Example 2.13 can be given as output. In the literature there is much work on learning BLPs and MLNs, cf. [26, 42]. But still, learning in the RME framework is still an open problem and part of current research. Therefore, up this moment KREATOR supports only learning of BLPs and MLNs.

Similar to the unified way of querying a knowledge base, the task of learning a knowledge base is addressed from within KREATOR in a most abstract fashion. Data sets can processed by KREATOR in form of sample files—which simply contain one ground atom per line—and can be fed to any learning algorithm made available by a developer in form of a plug-in. Consequently, a KREATOR plug-in which provides learning capabilities ensures that it transfers the ready-to-use sample data from KREATOR's data structure to its respective learning algorithm. A KREATOR plug-in may support multiple learning algorithms (so-called "learners") and different predefined configurations for each one. If one provides for a learning algorithm there has to be at least one so called "writer" which serializes the object structure of a knowledge base to a file in the specific file format. For example, the BLP implementation currently provides support for a writer which converts the learned BLP data structure to the Balios BLP syntax. Although this is the common syntax for BLPs, one might consider to implement another writer for an alternative BLP syntax. Figure 8 shows KREATOR's learning dialog which allows a direct selection of the favored learner and writer.

KREATOR's support for uniform sample files and its modular learner concept allows the user to perform learning experiments involving different approaches (and/or learners) in an easy and consistent way. In combination with KREATOR's comprehensive querying capabilities, this allows to conveniently compare the learning results across different approaches.

Currently the BLP and MLN plugins provide learning functionality. The learning algorithm for BLPs is implemented as a proof-of-concept and realizes a simple structure learner based on exhaustive search and uses a Maximum Likelihood parameter estimation, cf. [26]. The MLN plugin uses the learning algorithms provided by


Learn Knowledgebase

Learn a knowledge base from a sample collection.

Output file: learned_from_samples.blp

Learner: Standard learner for Bayesian Logic Programs

Writer: Balios BLP Syntax

Fig. 8. KReator – Learning dialog

Alchemy.

5 Examples

In this section, we present some more examples for probabilistic relational knowledge modeling. Each example has been modeled for all three knowledge representation formalisms and aims at (at least) one particular aspect of knowledge modeling. These aspects have especially been chosen to illustrate certain differences between the three formalisms. The modeling of an example in each formalism starts from the informal example description, i. e. we do not attempt to convert the modeling from one formalism to another one. We also give a short example that illustrates the learning functionalities (Section 5.3) and some runtime comparisons (Section 5.4). All examples have also been used during the development of KReator to evaluate several functionalities, especially those involved in the inference processes. We provide a whole set of examples in our model repository, accessible under `http://kreator.cs.tu-dortmund.de/`. The versions used for this paper can also be found under `http://kreator.cs.tu-dortmund.de/experiments/igpl2011.html`.

5.1 Birds and Penguins

In the following, we discuss the famous *Tweety* example which is often used in the context of non-monotonic reasoning. In this relational version of the example, we consider some birds and want to state some rules about their ability to fly. In particular, we want to state that birds typically fly, that penguins typically do not fly, and that every penguin is a bird. This example shows how well a formalism deals with conflicting information on exceptional individuals. Given a particular bird Tweety for which we have no knowledge of being a penguin we expect the formalism to derive with a high probability that Tweety does actually fly. Adding the information that Tweety is a penguin the formalism should derive that Tweety no longer flies.

Although Tweety is still a bird the more specific information that penguins do not fly shall override the general rule of birds flying.

Example 5.1
We first represent the example as a Bayesian logic program. Let *bird*/1, *penguin*/1, and *flies*/1 be predicates with $S(\mathit{bird}) = S(\mathit{penguin}) = S(\mathit{flies}) = \{\mathsf{true}, \mathsf{false}\}$. Then the above rules can be stated as the set $\{c_1, c_2, c_3\}$ of Bayesian clauses with

$$\begin{aligned} c_1 &: \quad (\mathit{bird}(\mathsf{X}) \mid \mathit{penguin}(\mathsf{X})) \\ c_2 &: \quad (\mathit{flies}(\mathsf{X}) \mid \mathit{bird}(\mathsf{X})) \\ c_3 &: \quad (\mathit{flies}(\mathsf{X}) \mid \mathit{penguin}(\mathsf{X})) \end{aligned}$$

and their corresponding conditional probability distributions $\{\mathsf{cpd}_{c_1}, \mathsf{cpd}_{c_2}, \mathsf{cpd}_{c_3}\}$ by

$$\begin{array}{cc} \mathsf{cpd}_{c_1}(\mathsf{true}, \mathsf{true}) = 1 & \mathsf{cpd}_{c_1}(\mathsf{false}, \mathsf{true}) = 0 \\ \mathsf{cpd}_{c_1}(\mathsf{true}, \mathsf{false}) = 0.5 & \mathsf{cpd}_{c_1}(\mathsf{false}, \mathsf{false}) = 0.5 \\ \mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{true}) = 0.9 & \mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{true}) = 0.1 \\ \mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{false}) = 0.2 & \mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{false}) = 0.8 \\ \mathsf{cpd}_{c_3}(\mathsf{true}, \mathsf{true}) = 0.01 & \mathsf{cpd}_{c_3}(\mathsf{false}, \mathsf{true}) = 0.99 \\ \mathsf{cpd}_{c_3}(\mathsf{true}, \mathsf{false}) = 0.3 & \mathsf{cpd}_{c_3}(\mathsf{false}, \mathsf{false}) = 0.7 \quad . \end{array}$$

Notice, that some of the probabilities defined for each conditional probability distribution are somewhat arbitrary. The problem is that defining a probability for a rule given that its premise is not fulfilled is a hard task. Considering clause c_2 saying that birds usually fly. But what is the probability of a non-bird flying? It is a serious drawback of Bayesian logic programs (and Bayes nets in general) that they demand a full specification of a conditional probability distribution when complete information is not available. This problem is resolved in logical Bayesian networks [10] by introducing purely logical statements that are not interpreted in a probabilistic sense. Furthermore, the Balios engine [28] for Bayesian logic programs allows the specification of logical predicates as well and a purely logical specification of background knowledge using an underlying Prolog theory. See [28, 27] for more details.

To complete the specification of the above Bayesian logic program we define the *noisy-or* function to be the combining rule for all predicates, cf. Example 2.4.

Example 5.2
To represent the scenario in Markov logic, we use implications to model the conditional knowledge from our example. As already discussed in the context of Example 2.9, the usage of conjunctions instead of implications might provide a better modelling. We also use the heuristic discussed in Example 2.9 to determine the weights of the formulas from the probabilities defined in the previous example. A much more accurate (but also more cumbersome) way would have been to generate a set of sample data which complies with these probabilities and to learn the weights from this sample data. Both points should be considered when evaluating the MLN results.

The weighted formulas for the Markov Logic Network are given by

$$2.1972 \qquad \mathit{bird}(\mathsf{x}) \Rightarrow \mathit{flies}(\mathsf{x}) \tag{5.1}$$
$$-4.5951 \qquad \mathit{penguin}(\mathsf{x}) \Rightarrow \mathit{flies}(\mathsf{x}) \tag{5.2}$$
$$\mathit{penguin}(\mathsf{x}) \Rightarrow \mathit{bird}(\mathsf{x}). \tag{5.3}$$

As said, notice that it is $2.1972 = \ln \frac{0.9}{0.1}$ and $-4.5951 = \ln \frac{0.01}{0.99}$. Notice further, that formula (5.3) has been given no weight at all. This is Alchemy syntax and equivalent to stating that formula (5.3) is a strict formula and should be considered with an infinite weight.

Example 5.3
From the approaches discussed in this paper the RME approach features the most concise and simple representation of knowledge. The Tweety example can be represented as a knowledge base $KB = \{r_1, r_2, r_3\}$ with

$$\begin{aligned} r_1 &: (\mathit{flies}(\mathsf{X}) \mid \mathit{bird}(\mathsf{X}))[0.9] \\ r_2 &: (\mathit{flies}(\mathsf{X}) \mid \mathit{penguin}(\mathsf{X}))[0.01] \\ r_3 &: (\mathit{bird}(\mathsf{X}) \mid \mathit{penguin}(\mathsf{X}))[1.0] \end{aligned}$$

This representation features both a simple declarative notation with a single attached value to each formula (likes Markov Logic Networks) and a declarative probabilistic semantics (like Bayesian Logic Programs).

Considering the Tweety example, the most interesting inferences one would like to draw concern the flying ability of some particular birds. Let our universe consist of the four birds Tweety, Huey, Dewey, Louie, and the penguin Opus (who is a bird as well). Therefore let the set of evidences be given by

$$\{\ \mathit{bird}(\mathsf{tweety}), \mathit{bird}(\mathsf{huey}), \mathit{bird}(\mathsf{dewey}), \mathit{bird}(\mathsf{louie}), \mathit{bird}(\mathsf{opus}), \mathit{penguin}(\mathsf{opus})\ \}$$

As one might expect, the inferences drawn on their flying ability concerning Tweety, Huey, Dewey, and Louie will be the same. For all these individuals we have represented the same knowledge (that they are birds) so they all will fly with the same probability. This phenomenon is called *prototypical indifference* and discussed in more detail in [45]. Hence, it suffices to consider what the different formalisms infer in the flying ability of one of these individuals (we chose Tweety). This reasoning does not apply for Opus who is a known penguin. So, we are also interested in the inferences on the flying ability drawn for Opus. Table 2 shows the results on the queries *flies*(tweety) and *flies*(opus) in the different formalisms. All probabilities are rounded off to three decimal places when necessary.

One thing to notice is that the inference concerning the flying ability of Opus drawn from the BLP differs significantly from the intended probability of 0.01. The inferred probability of 0.901 derives from the fact that both clauses c_2 and c_3 (see Example 5.1) are used when determining the probability of *flies*(opus) (as both apply for Opus) and combined using *noisy-or*. In contrast to this result the inference drawn on the flying ability of Tweety accurately resembles the intended probability when representing

	BLP	MLN	RME
$\mathit{flies}(\mathsf{tweety})$	0.900	0.102	0.900
$\mathit{flies}(\mathsf{opus})$	0.901	0.096	0.010

TABLE 2. Inferences for the Tweety example

the knowledge as a BLP. One can see that the selection of the "right" combining rule is crucial when representing knowledge using Bayesian logic programming. As a comparison, the inference on the flying ability of Opus drawn from the same BLP but using *noisy-and*[9] as the combining rule for *flies* yields a probability of 0.009; a much more adequate probability given the knowledge represented. However, consider a slight extension of the BLP given in Example 5.1 where we add the following clause:

$$c_4 \quad : \quad (\mathit{flies}(\mathsf{X}) \mid \mathit{rocket\_pack}(\mathsf{X}))$$

with

$$\mathsf{cpd}_{c_1}(\mathsf{true},\mathsf{true}) = 0.9 \qquad \mathsf{cpd}_{c_1}(\mathsf{false},\mathsf{true}) = 0.1$$
$$\mathsf{cpd}_{c_1}(\mathsf{true},\mathsf{false}) = 0.5 \qquad \mathsf{cpd}_{c_1}(\mathsf{false},\mathsf{false}) = 0.5$$

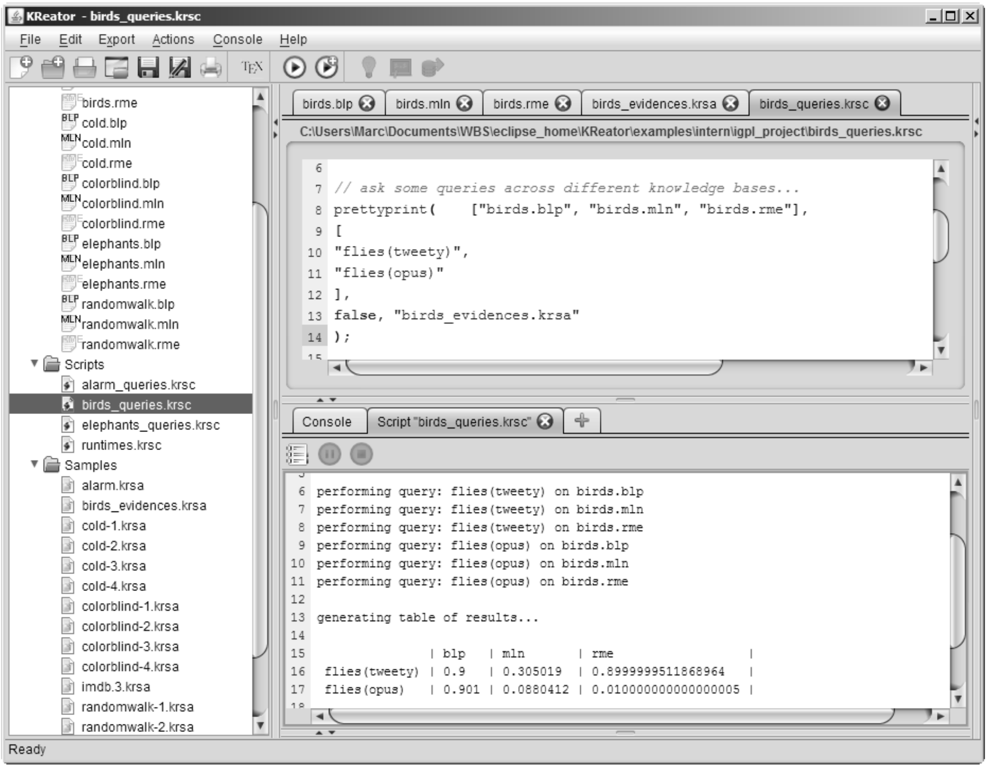
stating that the probability of a flying X is 0.9 when X has a rocket pack. Clearly, both being a bird and having a rocket pack are independent causes for the ability to fly and should strengthen the belief in the flying ability of X. But being a penguin should "override" the influence of being a bird on the ability to fly. As Bayesian logic programming allows only for a single combining rule to be assigned to a predicate this situation cannot be modeled without dropping the independence assumption. However, a naive incorporation of c_4 into the BLP yields a probabilistic model that—in our opinion—does not model the situation adequately. Considering a penguin without a rocket pack applying the combining rule should result in a small probability for flying while a penguin with a rocket pack should yield a high probability for flying. No combining rule can exhibit this behavior.

Complementary to the results drawn from the BLP the MLN gives an intuitive probability for the query $\mathit{flies}(\mathsf{opus})$ and an unexpected one for the query $\mathit{flies}(\mathsf{tweety})$. This stems from the fact that the default MLN inference algorithm (which has to be explicitly configured in Alchemy) makes no closed-world assumption. As there is no explicit knowledge given on whether Tweety is a penguin or not the formula (5.2) still has influence on the probability of $\mathit{flies}(\mathsf{tweety})$.

Lastly, the inferences drawn from the RME representation exactly reflects the represented knowledge. This stems from the excellent commonsensical properties of ME-inference, cf. [38, 24] and is (in this case) independent of the actually used grounding operator.

All the above computations were performed using KREATOR. Figure 9 shows a script in JAVASCRIPT syntax that concisely collects a set of queries to different representation formalisms for relational probabilistic reasoning. Having represented the

[9]The *noisy-and* combination of two probabilities p_1 and p_2 is defined as $p_1 \cdot p_2$. Remember that the *noisy-or* combination of two probabilities p_1 and p_2 is defined as $1 - (1 - p_1) \cdot (1 - p_2)$.



```

// ask some queries across different knowledge bases...
prettyprint(    ["birds.blp", "birds.mln", "birds.rme"],
[
"flies(tweety)",
"flies(opus)"
],
false, "birds_evidences.krsa"
);
```

```
performing query: flies(tweety) on birds.blp
performing query: flies(tweety) on birds.mln
performing query: flies(tweety) on birds.rme
performing query: flies(opus) on birds.blp
performing query: flies(opus) on birds.mln
performing query: flies(opus) on birds.rme

generating table of results...

               | blp   | mln       | rme                  |
 flies(tweety) | 0.9   | 0.305019  | 0.8999999511868964   |
 flies(opus)   | 0.901 | 0.0880412 | 0.010000000000000005 |
```

FIG. 9. A script with queries for the Tweety example and its output in the console

scenario in these formalisms and bundled the different knowledge bases in a project addressing these knowledge bases is simple and can be done in an unified manner. The script shown in Figure 9 is also included in our model repository so the results presented here can be reproduced and confirmed at any time.

5.2 Elephants and Keepers

This example was taken from [6] and has already been employed in Example 2.15 for discussing the problem of grounding relational conditional knowledge bases. The example under discussion describes a scenario in a zoo with elephants and their keepers. In general, it is very probable (90%) that an elephant likes a keeper. There is one special keeper, Fred, who has a strange sense of humor, so it is rather improbable (30%) that an elephant likes him. But there is also one special elephant, Clyde, who shares Fred's sense of humor, so Clyde likes Fred for sure (100%).

Thus, this example contains both default as well as specific knowledge about two types of individuals. From a pure logical point of view there is a conflict between the specific knowledge about Fred and the default knowledge about elephants and keepers. Since no explicit exception has been phrased the default knowledge also applies to Fred. But from a commonsensical point of view, it is obvious that the more specific knowledge about Fred should override the default knowledge about elephants and keepers, as well as the even more specific knowledge about Clyde and Fred should override the knowledge about elephants and Fred. Therefore it is interesting to compare how each of the three knowledge representation formalisms handles the problems arising from potential conflicts between default and specific knowledge.

In contrast to the other examples in this paper, the (uncertain) knowledge in this example is not conditioned, i. e. the example contains no rules but just facts. Hence, we avoid any side-effect introduced by the different ways of modeling if-then-rules (especially in the case of MLNs) and we can focus our comparison of the results on the handling of default and specific knowledge.

Example 5.4
We start by representing the above knowledge as a Bayesian logic program. We define $S(\mathit{likes}/2) = \{\mathsf{true}, \mathsf{false}\}$ with *noisy-or* as combining rule and a set $\{c_1, c_2, c_3\}$ of Bayesian clauses

$$\begin{aligned} c_1 &: \quad (\mathit{likes}(\mathsf{X}, \mathsf{Y})) \\ c_2 &: \quad (\mathit{likes}(\mathsf{X}, \mathsf{fred})) \\ c_3 &: \quad (\mathit{likes}(\mathsf{clyde}, \mathsf{fred})) \end{aligned}$$

with:

$$\begin{aligned} \mathsf{cpd}_{c_1}(\mathsf{true}) &= 0.90 & \mathsf{cpd}_{c_1}(\mathsf{false}) &= 0.10 \\ \mathsf{cpd}_{c_2}(\mathsf{true}) &= 0.30 & \mathsf{cpd}_{c_2}(\mathsf{false}) &= 0.70 \\ \mathsf{cpd}_{c_3}(\mathsf{true}) &= 1.00 & \mathsf{cpd}_{c_3}(\mathsf{false}) &= 0.00 \end{aligned}$$

Since there are no if-then-rules in this example, we can define the values of cpd_{c_i} straight forward, i. e. we do not have to deal with the problem of stating probabilities for non-fulfilled rules.

Example 5.5
We go on by modeling the scenario as a Markov logic network. We declare the types and respective constants $\mathit{elephant} = \{\mathsf{Clyde}, \mathsf{Dumbo}, \mathsf{Tuffi}\}$, $\mathit{keeper} = \{\mathsf{Fred}, \mathsf{Hank}\}$, the typed predicate $\mathit{likes}(\mathit{elephant}, \mathit{keeper})$ and the following weighted formulas:

$$\begin{aligned} 2.1972 &\quad \mathit{likes}(\mathsf{x}, \mathsf{y}) \\ -0.8573 &\quad \mathit{likes}(\mathsf{x}, \mathsf{Fred}) \\ &\quad \mathit{likes}(\mathsf{Clyde}, \mathsf{Fred}). \end{aligned}$$

Formula $\mathit{likes}(\mathsf{Clyde}, \mathsf{Fred})$ has no weight assigned explicitly because it is a strict formula (with an implicitly infinite weight). The weights of the other formulas have been determined by the heuristic described in Example 2.9, i. e. $2.1972 = \log \frac{0.9}{0.1}$ and $-0.8573 = \log \frac{0.3}{0.7}$.

Example 5.6
Finally, we recast the representation of the scenario in the RME framework, cf. Example 2.15. We declare the types and respective constants $\mathit{Elephant} = \{\mathsf{clyde}, \mathsf{dumbo}, \mathsf{tuffi}\}$, $\mathit{Keeper} = \{\mathsf{fred}, \mathsf{hank}\}$, the typed predicate $\mathit{likes}(\mathit{Elephant}, \mathit{Keeper})$ and the following conditionals:

	BLP	MLN	RME
$\mathit{likes}(\mathsf{dumbo}, \mathsf{hank})$	0.900	0.897	0.900
$\mathit{likes}(\mathsf{clyde}, \mathsf{hank})$	0.900	0.897	0.900
$\mathit{likes}(\mathsf{dumbo}, \mathsf{fred})$	0.930	0.808	0.300
$\mathit{likes}(\mathsf{clyde}, \mathsf{fred})$	1.000	1.000	1.000

TABLE 3. Inferences for the Elephants-and-Keepers example

$$\begin{aligned}(\mathit{likes}(\mathsf{X}, \mathsf{Y})) &\quad [0.9]\\ (\mathit{likes}(\mathsf{X}, \mathsf{fred})) &\quad [0.3]\\ (\mathit{likes}(\mathsf{clyde}, \mathsf{fred})) &\quad [1.0]\end{aligned}$$

Note that in order to handle the above knowledge base appropriately with the RME framework a more sophisticated grounding operator has to be used that solves the conflicts between the contradictory statements involving Clyde or Fred. Here, we use the *specificity* grounding operator which prefers instances of more specific conditionals over instances of less specific conditionals. For example, groundings of the more specific conditional $(\mathit{likes}(\mathsf{X}, \mathsf{fred}))[0.3]$ are preferred over groundings of the less specific conditional $(\mathit{likes}(\mathsf{X}, \mathsf{Y}))[0.9]$; but the (already ground) conditional $(\mathit{likes}(\mathsf{clyde}, \mathsf{fred}))[1.0]$ is preferred over the conflicting grounding of the (comparatively) less specific conditional $(\mathit{likes}(\mathsf{X}, \mathsf{fred}))[0.3]$. That way, the *specificity* grounding operator keeps only instances of conditionals that are most specific, cf. [33] for a more detailed elaboration.

To analyze the different modelings and their respective handling of conflicting default and specific knowledge, we formulate the queries stated in the left column of Table 3 and infer their respective probabilities from each knowledge representation formalism. The queries cover the four kinds of interesting relations of elephants with keepers: an ordinary elephant (Dumbo) with an ordinary keeper (Hank), the special elephant Clyde with an ordinary keeper, an ordinary elephant with the special keeper Fred, as well as the special elephant Clyde with the special keeper Fred. The results show that each formalism infers, as expected, the same probability for $\mathit{likes}(\mathsf{dumbo}, \mathsf{hank})$ and $\mathit{likes}(\mathsf{clyde}, \mathsf{hank})$. That is, although Clyde is an exceptional elephant considering its relationship with Fred, this does not distinguish Clyde from any other elephant (e. g. Dumbo) in its relationship to an ordinary keeper (e. g. Hank).

The inferred BLP results show that the default knowledge is represented as intended, whereas the more specific knowledge of c_2 has not been handled properly. For the query $\mathit{likes}(\mathsf{dumbo}, \mathsf{fred})$ both clauses c_1 and c_2 are applicable and their probabilities are strengthened by the selected combining rule. As already mentioned in analysis of Example 5.1 this is due to the chosen combining rule *noisy-or* for the predicate *likes*. Performing the same calculations with *noisy-and* as combining function, we get the results 0.900 / 0.900 / 0.27 / 0.27. So, *noisy-and* handles the specific knowledge about Dumbo and Fred as intended, but results in a probability for $\mathit{likes}(\mathsf{clyde}, \mathsf{fred})$ which is almost inverse to the intended probability.

The results inferred from the MLN approximate the default knowledge quite well and the specific knowledge about Clyde and Fred is represented as intended. But the probability inferred for the relationship between an ordinary elephant and Fred completely fails the intended probability. This is due to the heuristically determined weights, which do not represent the relative importance of the respective formula adequately in this case. In order to decrease the inferred probability of $\mathit{likes}(\mathsf{dumbo}, \mathsf{fred})$, it can be presumed that decreasing the weight of formula $\mathit{likes}(\mathsf{X}, \mathsf{fred})$ might have the desired effect, i. e. that a world satisfying this formula will suffer a stronger decrease in probability. In fact, adjusting the formula's weight to -3.1 has the desired effect: We infer the probability 0.302 for the query $\mathit{likes}(\mathsf{dumbo}, \mathsf{fred})$, whereas all other (rounded) probabilities stay the same. So it becomes clear that, in principle, MLNs can handle conflicts between default and specific knowledge quite well, but the determination of appropriate weights is crucial for an adequate knowledge representation and inference. In this simple example without complex dependencies between formulas, guessing an appropriate weight was still feasible. But in slightly more complex examples this is hardly possible, so that appropriate weights can only be determined by learning from sample data.

The query results inferred from the RME match the intended probabilities perfectly. Thus, the RME approach can handle the default and the specific knowledge most directly. This is due to the chosen grounding operator in this example, which deals with conflicting parts of the knowledge exactly as intended, and the characteristics of ME-optimal knowledge representation, which represents the knowledge as unbiasedly as possible, and therefore prevents any unintended effects in the inference process.

Note, that both, the BLP from Example 5.4 and the MLN from Example 5.5 can be modified in order to account for the exceptional individuals Clyde and Fred. For example, the MLN from Example 5.5 can be rewritten as

$$\begin{array}{rl} 2.1972 & \mathit{likes}(\mathsf{x}, \mathsf{y}) \wedge y \neq Fred \\ -0.8573 & \mathit{likes}(\mathsf{x}, \mathsf{Fred}) \wedge x \neq Clyde \\ & \mathit{likes}(\mathsf{Clyde}, \mathsf{Fred}) \end{array}$$

Querying the above MLN with the queries from Table 3 results in much more adequate inferences than the original MLN: 0.884 / 0.876 / 0.283 / 1.000. However, imagine that the knowledge captured in the MLN of Example 5.5 is distributed over several sources and only combined during runtime. One source might maintain general knowledge on the domain (such as the first formula) and other sources might contribute with specific information on individuals (such as the other formulas). In this scenario a representation like the one above cannot be realized. Therefore, representing the knowledge with the modified MLN above hinders modularity and also makes extending the knowledge base with new information a hard task. A similar discussion applies to the BLP as well.

All results from this example have been calculated with KReator by simply performing the execution of a single JavaScript file which contained all queries. As the above discussion of the results pointed out, for a knowledge engineer it is often necessary to experiment with certain aspects a knowledge base (e. g. combining rules, or weights) or the inference process. In such situations, KReator's scripting abilities prove to be very helpful, since the execution of a JavaScript file allows the convenient recalculation of all results after some modifications have been made to a knowledge base. Even more, important parameters of the inference process (e. g. the RME grounding operator) can be set directly within the JavaScript file. Thus, such parameters can easily be changed and the script be rerun to compare the results and therefore analyze the effects of the respective inference parameter.

5.3 Learning about Birds and Penguins

While the previous examples were concerned with reasoning tasks we give now a small example for learning knowledge bases from data. As there is, up to now, no learning algorithm for RME available (this is an open research problem and left for future work) we will concentrate on the BLP and MLN formalisms.

We use the same scenario as in Section 5.1 and employ a very useful functionality of KReator: data generation from knowledge bases. This functionality allows to generate sample data from some knowledge base that describes the knowledge base with respect to the knowledge bases's rules and parameters. We omit a more detailed description of this functionality as it depends on the specific approach. As a starting point we use the BLP from Example 5.1 from which we generate 500 data sets. Each of these data sets consists of expressions of the form

$$\begin{aligned} \mathit{penguin}(\mathsf{a}) &= B \\ \mathit{bird}(\mathsf{a}) &= B \\ \mathit{flies}(\mathsf{a}) &= B \end{aligned}$$

with some constant a and $B \in \{\mathsf{true}, \mathsf{false}\}$. The statistical distribution of birds that are penguins and/or fly resembles the probabilities in the CPD of the BLP in Example 5.1.

Applying the standard learning algorithm for BLPs without optimization onto the sample data results in a new BLP given via

$$\begin{aligned} c_1 &: (\mathit{flies}(\mathsf{X}) \mid \mathit{bird}(\mathsf{X})) \\ c_2 &: (\mathit{penguin}(\mathsf{X}) \mid \mathit{bird}(\mathsf{X})) \\ c_3 &: (\mathit{bird}(\mathsf{X})) \end{aligned}$$

with corresponding conditional probability distributions $\{\mathsf{cpd}_{c_1}, \mathsf{cpd}_{c_2}, \mathsf{cpd}_{c_3}\}$ given via

$$\begin{aligned} \mathsf{cpd}_{c_1}(\mathsf{true},\mathsf{true}) &= 0.475 & \mathsf{cpd}_{c_1}(\mathsf{false},\mathsf{true}) &= 0.525 \\ \mathsf{cpd}_{c_1}(\mathsf{true},\mathsf{false}) &= 0.319 & \mathsf{cpd}_{c_1}(\mathsf{false},\mathsf{false}) &= 0.681 \\ \mathsf{cpd}_{c_2}(\mathsf{true},\mathsf{true}) &= 0.674 & \mathsf{cpd}_{c_2}(\mathsf{false},\mathsf{true}) &= 0.326 \\ \mathsf{cpd}_{c_2}(\mathsf{true},\mathsf{false}) &= 0 & \mathsf{cpd}_{c_2}(\mathsf{false},\mathsf{false}) &= 1 \\ \mathsf{cpd}_{c_3}(\mathsf{true}) &= 0.774 & \mathsf{cpd}_{c_3}(\mathsf{false}) &= 0.226 \quad . \end{aligned}$$

Notice, that the above BLP differs significantly from the BLP in Example 5.1, both in structure and probabilities. While the differences in the structure of the BLPs stem from the employed learning algorithm resp. the parameters of this algorithm the differences of the parameters—e. g. the lower probability of a bird that flies—come from the underlying population in the sample data. As one can see from the values of cpd_{c_2} nearly two thirds of the birds considered are penguins and as a consequence the probability of a flying bird is considerably lower. This is due to the fact that the BLP in Example 5.1 gives no information on the distribution of penguins among the birds and thus the same holds for the generated data.

Applying the default beam search structure learning algorithm for MLNs [30] without optimization results in the MLN given via

$$\begin{array}{rl} 0.110111 & \mathit{bird}(\mathsf{x}) \\ -0.238245 & \mathit{flies}(\mathsf{x}) \\ 0.728441 & \mathit{penguin}(\mathsf{x}) \\ 7.00987 & \neg\mathit{penguin}(\mathsf{x}) \vee \mathit{bird}(\mathsf{x}) \end{array}$$

The main remark to be made to this MLN is that no relationship between the attributes *bird*, *penguin*, *flies* have been found except for the quite obvious rule that penguins are birds. Although the MLN above describes the sample data very well it is not very informative as there is no relationship given for *bird* and *flies*. However, this stems from the fact that the learning algorithm used in this experiment is the default learning algorithm of Alchemy without parameter optimization.

As can be seen from the above discussion the two complementary functionalities of generating data and learning are a helpful means to understand knowledge represented in some knowledge base.

5.4 Runtime analysis

Table 4 shows a runtime comparison of the current reasoner implementations for BLPs, MLNs, and RME. When comparing the given runtimes one has to be aware of the fact that the current implementations of the reasoners for Bayesian logic programming and relational maximum entropy are experimental and mainly serve as an illustration for KREATOR's functionalities. The current implementation of the BLP reasoner uses a very naive approach for (exact) inference by first building the SLD tree for some query and then propagating probabilities down to the query atom in question. For future work we plan to integrate a propositional Bayesian network reasoner such as the *Bayesian Network tools for Java*[10] to allow for faster and approximate inference. The current implementation of the RME approach relies heavily on the underlying propositional maximum entropy reasoner, cf. Section 3.3. Most of these reasoners are hardly applicable for large knowledge bases due to memory insufficiencies. In Table 4, cells with a dash indicate a memory error when trying to query the RME knowledge base with the given evidences. As approximate inference is an open problem for relational maximum entropy further investigations have to be made before implementing a scalable version of the RME inference mechanism. Only the MLN implementation in KREATOR uses a mature tool for processing its knowledge bases: Alchemy. This application has been developed for quite some time, so it is not surprising that it features good running times as given in Table 4. However, one has to note, that Alchemy employs approximate inference only and that the above runtimes stem from running Alchemy with the default setting. The experiments measure inference behavior on

[10]`http://bnj.sourceforge.net/`

queries and were performed on three different knowledge bases with increasingly complex evidences. They were performed on an Intel Core 2 Duo with 3.06 GHz and 4 GB of memory and Table 4 shows the average runtime out of 10 runs.

In Table 4, the knowledge base `colorblind` describes a genetic model of inheriting color blindness from ancestors. It assumes some pedigree given as evidence and describes how color blindness inherits from the roots—ancestors with no parents given—down to other persons in the pedigree. The experiments `colorblind-1` to `colorblind-4` use the same knowledge base but with an increasing size of the given pedigree (with 5, 21, 41, 101 persons, respectively). The knowledge base `cold` describes a probabilistic model of cold contagion within some population. The rules in `cold` describe some initial probability of having a cold, the probability of having a cold if one is susceptible, and the probability of having a cold if one had contact with someone who has a cold. The experiments `cold-1` to `cold-4` use the same knowledge base but with an increasing size of the underlying population (with 5, 10, 20, 50 persons, respectively). The knowledge base `randomwalk` describes a probabilistic model of a simple random walk. Given two positions and a probability of 0.8 for switching position the rules in `randomwalk` describe the progress over some time steps. As before, the experiments `randomwalk-1` to `randomwalk-4` use the same knowledge base but increasingly consider more time steps (5, 10, 20, 50 time steps, respectively).

For each formalism a separate representation of each knowledge base has been used that model the same knowledge in an approximate manner. A complete listing of the knowledge bases used in the runtime analysis can be found in Appendix A. Note, that the knowledge base `cold` cannot be represented as a BLP in a direct fashion because the *contact* relation used to determine one's probability of having a cold is a symmetric relation that may introduce cycles into the ground Bayesian network. Consider the following excerpt from `cold.rme`:

$$(\mathit{cold}(\mathsf{X}) \mid \mathit{contact}(\mathsf{X},\mathsf{Y}), \mathit{cold}(\mathsf{Y}), \mathsf{X} \neq \mathsf{Y})[0.6]$$

This clause means that the probability of X having a cold depends on the probability of Y having a cold if X had contact with Y. Here, *contact* is meant to be symmetric, so the probability of, say, Jack has a cold depends on the probability of, say, Christine, having a cold which itself depends on Jack having a cold (if both had contact with each other). This kind of relationship cannot be modeled in a BLP in general, cf. Section 2.1. In order to apply Bayesian logic programming to the `cold` example the data used in the experiments reflected in Table 4 for `cold` contains no cycles for *contact* and the BLP itself does not perform a "symmetric closure" on *contact*, i. e., while the MLN for `cold` contains the strict formula

$$\mathit{contact}(\mathsf{X},\mathsf{Y}) \Leftrightarrow \mathit{contact}(\mathsf{Y},\mathsf{X}).$$

the BLP does not contain something similar.

All data used for undertaking the presented experiments can be found under `http://kreator.cs.tu-dortmund.de/experiments/igpl2011.html` in form of a KReator project. This project also contains a script directly executable from within KReator that runs all queries and measures the runtimes automatically.

	BLP	MLN	RME
colorblind-1	0.069	0.117	9157.561
colorblind-2	2.314	0.230	–
colorblind-3	33.897	0.273	–
colorblind-4	5573.572	0.459	–
cold-1	0.015	0.189	0.438
cold-2	0.023	0.189	–
cold-3	0.079	0.264	–
cold-4	1.667	0.820	–
randomwalk-1	0.020	0.167	0.174
randomwalk-2	0.023	0.180	4223.182
randomwalk-3	0.078	0.192	–
randomwalk-4	0.996	0.269	–

TABLE 4: Runtime comparison of the different implementations (all values are given in seconds).

6 Summary and Future Work

In this paper we presented KREATOR and illustrated its system architecture and usage. Although KREATOR is still in an early stage of development it already supports Bayesian logic programs, Markov logic networks, and relational maximum entropy. For each of these formalisms we provided an extensive overview and illustrated their differences using several examples. By doing so, we also showed the advantages of using a user-friendly toolbox like KREATOR for comparing and evaluating different formalisms for relational probabilistic reasoning.

While their exist many prototypical implementations of specific approaches, non of these systems follows KREATOR's approach in providing a general and unifying framework for *different* approaches to statistical relational learning or inductive logic programming. To our knowledge, there is only one software system which takes an approach comparable to KREATOR in the sense that it combines different formalisms within one system. The *ProbCog* (Probabilistic Cognition for Cognitive Technical Systems) system[11] [23] is a software suite for statistical relational learning and currently supports three different knowledge representation approaches. But the primary application focus of the ProbCog system differs significantly from KREATOR, since ProbCog is developed for its intended practical application and integration in cognitive technical systems. So it focuses on providing a versatile and efficient framework for that specific purpose and does not provide a unified graphical user interface, whereas KREATOR's focus is on facilitating the typical workflow of a knowledge engineer, researcher, or developer.

Although KREATOR is now at a usable state for many scientific tasks in the area of relational probabilistic reasoning there are still a lot of plans on future development,

[11] http://ias.cs.tum.edu/research-areas/knowledge-processing/probcog

most notable of these being the integration of learning knowledge bases from data. Due to the open architecture of KReator and the ability to perform many tasks on abstract notions of e. g. knowledge bases the task of implementing learning algorithms for different representation formalisms will benefit from many commonalities of these algorithms. Most approaches on learning statistical relational models from data rely on established work from propositional learners. Learning the structure of relational probabilistic models can be done using standard inductive logic programming systems like CLAUDIEN [40] or MACCENT [5]. Learning the values (probabilities) of the models can be performed using e. g. the EM-algorithm (expectation maximization, see [7]). These common components will certainly simplify implementing the ability to learn different knowledge bases from data within KReator. In order to gain the ability to learn knowledge bases for RME (which differs significantly from other relational models which mostly rely on graphical notions and probabilistic dependence/independence assumptions) we plan to integrate an extended version of the CondorCKD system [15, 25]. CondorCKD is a propositional learning system for conditionals that relies on an algebraic characterization of interrelationships between conditionals in a knowledge base, cf. [24].

We plan to enhance KReator's unified query syntax to allow more complex queries. This requires more sophisticated conversion patterns to translate a unified query to the respective target syntax, e. g. to handle multi-state BLP predicates in an automated way. The enhancement of the query syntax will go along with the development of an even more challenging feature: We plan on introducing some kind of unified knowledge base (template) format. The final goal is to be able to formulate (at least) the central aspects of a knowledge base in a unified syntax and to have this knowledge base be converted to different target languages more or less automatically. Having this functionality available will further improve the handling and comparison of different knowledge representation formalisms. In order to evaluate this feature on a large scale we also plan to implement other formalisms for relational probabilistic knowledge representation. In particular, we plan to add support for logical Bayesian networks [9] and probabilistic relational models [18], as well as to use KReator as a testbed to evaluate other approaches for relational probabilistic reasoning under maximum entropy [45].

We also plan to introduce JavaScript commands which shall allow to modify central aspects of a knowledge base. For example, one command could permit to "disable" a rule of the knowledge base, so that the impact of this rule to a certain query result could be analyzed. Another command could enable the user to alter a rule's quantification value (probability or weight) successively to observe its impact to a certain query.

Since one central feature of KReator is to support the user when comparing query results of different knowledge bases, we will continue to extend this functionality in future development. For example, a future release of KReator will present the results of queries addressing different knowledge bases in a spreadsheet-like structure. That way, the user will be able to instantly compare all inferred probabilities at a glance.

KReator is available under the GNU General Public License and the latest version can be obtained from `http://kreator.cs.tu-dortmund.de`.

Acknowledgements. The research reported here was partially supported by the Deutsche Forschungsgemeinschaft (grants BE 1700/7-1 and KE 1413/2-1). We also thank the anonymous reviewers for their helpful suggestions to improve this paper and Sebastian Loh for implementing most of the functionalities in KReator and the initial work on relational maximum entropy.

References

[1] Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic Reasoning with Answer Sets. *Theory and Practice of Logic Programming*, 2009.

[2] John S. Breese. Construction of Belief and Decision Networks. *Computational Intelligence*, 8(4):624–647, 1992.

[3] James Cussens. Logic-based Formalisms for Statistical Relational Learning. In Lise Getoor and Ben Taskar, editors, *An Introduction to Statistical Relational Learning.* MIT Press, 2007.

[4] Luc De Raedt and Kristian Kersting. Probabilistic Inductive Logic Programming. In Luc De Raedt, Kristian Kersting, Niels Landwehr, Stephen Muggleton, and Jianzhong Chen, editors, *Probabilistic Inductive Logic Programming*, pages 1–27. Springer, 2008.

[5] Luc Dehaspe. Maximum Entropy Modeling with Clausal Constraints. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 109–125. Springer, 1997.

[6] James P. Delgrande. On First-Order Conditional Logics. *Artificial Intelligence*, 105(1–2):105–137, 1998.

[7] Arthur P. Dempster, Laird. Nan M., and Donald B. Rubin. Maximum-Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 1977.

[8] Pedro Domingos and Daniel Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.

[9] Daan Fierens. *Learning Directed Probabilistic Logical Models from Relational Data.* PhD thesis, Katholieke Universiteit Leuven, 2008.

[10] Daan Fierens, Hendrik Blockeel, Jan Ramon, and Maurice Bruynooghe. Logical Bayesian Networks. In S. Dzeroski and H. Blockeel, editors, *Proceedings of the 3nd International Workshop on Multi-Relational Data Mining*, pages 19–30, 2004.

[11] Marc Finthammer, Christoph Beierle, Benjamin Berger, and Gabriele Kern-Isberner. Probabilistic Reasoning at Optimum Entropy with the MEcore System. In *Proceedings of the 22nd International FLAIRS Conference (FLAIRS'09).* AAAI Press, 2009.

[12] Marc Finthammer, Sebastian Loh, and Matthias Thimm. Towards a Toolbox for Relational Probabilistic Knowledge Representation, Reasoning, and Learning. In *Workshop on Relational Approaches to Knowledge Representation and Learning, Proceedings*, pages 34–48, Paderborn, Germany, September 2009.

[13] Jens Fisseler. Toward Markov Logic with Conditional Probabilities. In David C. Wilson and H. Chad Lane, editors, *Proceedings of the 21st International FLAIRS Conference*, pages 643–648. AAAI Press, 2008.

[14] Jens Fisseler. First-Order Probabilistic Conditional Logic: Introduction and Representation. In *Workshop on Relational Approaches to Knowledge Representation and Learning at KI-2009, Proceedings*, Paderborn, Germany, September 2009.

[15] Jens Fisseler, Gabriele Kern-Isberner, Christoph Beierle, Andreas Koch, and Christian Müller. Algebraic Knowledge Discovery using Haskell. In *Practical Aspects of Declarative Languages, 9th International Symposium.* Springer, 2007.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[17] Michael Gelfond and Nicola Leone. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 138(1–2):3–38, June 2002.

[18] Lise Getoor, Nir Friedman, Daphne Koller, and Benjamin Tasker. Learning Probabilistic Models of Relational Structure. In C. E. Brodley and A. P. Danyluk, editors, *Proceedings of the 18th International Conference on Machine Learning (ICML 2001).* Morgan Kaufmann, 2001.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[20] Adam J. Grove, Joseph Y. Halpern, and Daphne Koller. Random worlds and maximum entropy. *Journal of Artificial Intelligence Research (JAIR)*, 2:33–88, 1994.

[21] Manfred Jaeger. Relational Bayesian Networks: A Survey. *Electronic Transactions in Artificial Intelligence*, 6, 2002.

[22] Manfred Jaeger. Model-Theoretic Expressivity Analysis. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming: Theory and Application*, volume 4911 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 2008.

[23] Dominik Jain, Lorenz Mösenlechner, and Michael Beetz. Equipping Robot Control Programs with First-Order Probabilistic Reasoning Capabilities. In *International Conference on Robotics and Automation (ICRA)*, pages 3130–3135, 2009.

[24] Gabriele Kern-Isberner. *Conditionals in nonmonotonic reasoning and belief revision*. Number 2087 in Lecture Notes in Computer Science. Springer, 2001.

[25] Gabriele Kern-Isberner and Jens Fisseler. Knowledge Discovery by Reversing Inductive Knowledge Representation. In *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR-2004*, pages 34–44. AAAI Press, 2004.

[26] Kristian Kersting and Luc De Raedt. Basic Principles of Learning Bayesian Logic Programs. Technical report 174, Institute for Computer Science, University of Freiburg, Germany, 2002.

[27] Kristian Kersting and Luc De Raedt. Bayesian Logic Programming: Theory and Tool. In Lise Getoor and Ben Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

[28] Kristian Kersting and Uwe Dick. Balios – The Engine for Bayesian Logic Programs. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD-2004)*, pages 549–551, Sepember 2004.

[29] Nikhil S. Ketkar, Lawrence B. Holder, and Diane J. Cook. Comparison of Graph-based and Logic-based Multi-Relational Data Mining. *SIGKDD Explorations Newsletter*, 7(2):64–71, 2005.

[30] Stanley Kok and Pedro Domingos. Learning the structure of Markov logic networks. In *Proceedings of the 22nd international conference on Machine learning (ICML'05)*, pages 441–448. ACM Press, 2005.

[31] Stanley Kok, Parag Singla, Matthew Richardson, Pedro Domingos, Marc Sumner, Hoifung Poon, Daniel Lowd, and Jue Wang. *The Alchemy System for Statistical Relational AI: User Manual*. Department of Computer Science and Engineering, University of Washington, 2008.

[32] Vladimir Lifschitz. Foundations of Logic Programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, 1996.

[33] Sebastian Loh, Matthias Thimm, and Gabriele Kern-Isberner. On the Problem of Grounding a Relational Probabilistic Conditional Knowledge Base. In *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR'10)*, Toronto, Canada, May 2010.

[34] Thomas Lukasiewicz and Gabriele Kern-Isberner. Probabilistic Logic Programming under Maximum Entropy. In *Proceedings of the Fifth European Conferences on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-99)*, pages 279–292, 1999.

[35] Stephen Muggleton and Jianzhong Chen. A Behavioral Comparison of some Probabilistic Logic Models. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming: Theory and Application*, volume 4911 of *Lecture Notes in Computer Science*, pages 305–324. Springer, 2008.

[36] Stephen H. Muggleton. Stochastic logic programs. In Luc de Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–?264. IOS Press, Amsterdam, Netherlands, 1996.

[37] Donald Nute and Charles Cross. Conditional logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 4, pages 1–98. Kluwer Academic Publishers, second edition edition, 2002.

[38] Jeff Paris. *The Uncertain Reasoner's Companion – A Mathematical Perspective*. Cambridge University Press, 1994.

[39] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1998.

[40] Luc De Raedt and Luc Dehaspe. Clausal Discovery. *Machine Learning*, 26:99–146, 1997.

[41] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A Probabilistic Prolog and its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2462–2467, Hyderabad, India, 2007.

[42] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1–2):107–136, 2006.

[43] Wilhelm Rödder. Conditional logic and the principle of entropy. *Artificial Intelligence*, 117:83–106, 2000.

[44] Wilhelm Rödder and Carl-Heinz Meyer. Coherent Knowledge Processing at Maximum Entropy by SPIRIT. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI 1996)*, pages 470–476, 1996.

[45] Matthias Thimm and Gabriele Kern-Isberner. On probabilistic inference in relational conditional logics. *This volume*, 2010.

[46] Michael P. Wellman, John S. Breese, and Robert P. Goldman. From Knowledge Bases to Decision Models. *The Knowledge Engineering Review*, 7(1):35–53, 1992.

A Further Examples

In the following, formalizations of the knowledge bases used in Section 5.4 are given. For reasons of clarity of presentation we omit giving the specification of the conditional probability distributions for BLPs. These and all knowledge bases presented in this paper can be found at `http://kreator.cs.tu-dortmund.de/experiments/igpl2011.html`.

A.1 BLP *formalization of* `colorblind`

$$
\begin{aligned}
&(\mathit{gen\_carrier}(\mathsf{A}) \mid \mathit{male}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \mathit{gen\_carrier}(\mathsf{B}), \mathit{color\_blind}(\mathsf{B}))\\
&(\mathit{color\_blind}(\mathsf{A}) \mid \mathit{male}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \mathit{gen\_carrier}(\mathsf{B}), \mathit{color\_blind}(\mathsf{B}))\\
&(\mathit{color\_blind}(\mathsf{A}) \mid \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \mathit{parent}(\mathsf{C},\mathsf{A}), \mathit{gen\_carrier}(\mathsf{B}),\\
&\qquad\qquad \mathit{gen\_carrier}(\mathsf{C}), \mathit{color\_blind}(\mathsf{C}))\\
&(\mathit{gen\_carrier}(\mathsf{A}) \mid \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \mathit{parent}(\mathsf{C},\mathsf{A}), \mathit{gen\_carrier}(\mathsf{B}),\\
&\qquad\qquad \mathit{gen\_carrier}(\mathsf{C}), \mathit{color\_blind}(\mathsf{C}))
\end{aligned}
$$

A.2 MLN *formalization of* `colorblind`

$$
\begin{aligned}
& &&\neg\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \mathit{color\_blind}(\mathsf{B}) \wedge \mathsf{A} \neq \mathsf{B}\\
& &&\qquad \Rightarrow \mathit{gen\_carrier}(\mathsf{A})\\
& &&\neg\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \mathit{color\_blind}(\mathsf{B}) \wedge \mathsf{A} \neq \mathsf{B}\\
& &&\qquad \Rightarrow \mathit{color\_blind}(\mathsf{A})\\
&-1 &&\neg\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \neg\mathit{color\_blind}(\mathsf{B}) \wedge \mathsf{A} \neq \mathsf{B}\\
& &&\qquad \Rightarrow \mathit{gen\_carrier}(\mathsf{A})\\
&-1 &&\neg\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \neg\mathit{color\_blind}(\mathsf{B}) \wedge \mathsf{A} \neq \mathsf{B}\\
& &&\qquad \Rightarrow \mathit{color\_blind}(\mathsf{A})\\
& &&\neg\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \neg\mathit{gen\_carrier}(\mathsf{B}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \neg\mathit{gen\_carrier}(\mathsf{A})\\
& &&\neg\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \neg\mathit{gen\_carrier}(\mathsf{B}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \neg\mathit{color\_blind}(\mathsf{A})\\
& &&\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \mathit{parent}(\mathsf{C},\mathsf{A}) \wedge \mathit{gen\_carrier}(\mathsf{C})\\
& &&\qquad \wedge \mathit{color\_blind}(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \Rightarrow \mathit{gen\_carrier}(\mathsf{A})\\
& &&\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \mathit{parent}(\mathsf{C},\mathsf{A}) \wedge \mathit{gen\_carrier}(\mathsf{C})\\
& &&\qquad \wedge \mathit{color\_blind}(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \Rightarrow \mathit{color\_blind}(\mathsf{A})\\
& &&\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \mathit{parent}(\mathsf{C},\mathsf{A}) \wedge \mathit{gen\_carrier}(\mathsf{C})\\
& &&\qquad \wedge \neg\mathit{color\_blind}(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \Rightarrow \mathit{gen\_carrier}(\mathsf{A})\\
&-1 &&\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \mathit{parent}(\mathsf{C},\mathsf{A}) \wedge \mathit{gen\_carrier}(\mathsf{C})\\
& &&\qquad \wedge \neg\mathit{color\_blind}(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \Rightarrow \mathit{color\_blind}(\mathsf{A})\\
& &&\mathit{female}(\mathsf{A}) \wedge \mathit{parent}(b,a) \wedge \mathit{gen\_carrier}(\mathsf{B}) \wedge \mathit{parent}(\mathsf{C},\mathsf{A}) \wedge \neg\mathit{gen\_carrier}(\mathsf{C})\\
& &&\qquad \wedge \neg\mathit{color\_blind}(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \Rightarrow \mathit{gen\_carrier}(\mathsf{A})
\end{aligned}
$$

$$female(\mathsf{A}) \wedge parent(b,a) \wedge gen\_carrier(\mathsf{B}) \wedge parent(\mathsf{C},\mathsf{A}) \wedge \neg gen\_carrier(\mathsf{C}) \\ \wedge \neg color\_blind(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \wedge \neg color\_blind(\mathsf{A})$$

$$female(\mathsf{A}) \wedge parent(b,a) \wedge \neg gen\_carrier(\mathsf{B}) \wedge parent(\mathsf{C},\mathsf{A}) \wedge gen\_carrier(\mathsf{C}) \\ \wedge color\_blind(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \Rightarrow gen\_carrier(\mathsf{A})$$

$$female(\mathsf{A}) \wedge parent(b,a) \wedge \neg gen\_carrier(\mathsf{B}) \wedge parent(\mathsf{C},\mathsf{A}) \wedge gen\_carrier(\mathsf{C}) \\ \wedge color\_blind(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \wedge \neg color\_blind(\mathsf{A})$$

-1

$$female(\mathsf{A}) \wedge parent(b,a) \wedge \neg gen\_carrier(\mathsf{B}) \wedge parent(\mathsf{C},\mathsf{A}) \wedge gen\_carrier(\mathsf{C}) \\ \wedge \neg color\_blind(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \Rightarrow gen\_carrier(\mathsf{A})$$

$$female(\mathsf{A}) \wedge parent(b,a) \wedge \neg gen\_carrier(\mathsf{B}) \wedge parent(\mathsf{C},\mathsf{A}) \wedge gen\_carrier(\mathsf{C}) \\ \wedge \neg color\_blind(\mathsf{C}) \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \wedge \neg color\_blind(\mathsf{A})$$

$$female(\mathsf{A}) \wedge parent(b,a) \wedge \neg gen\_carrier(\mathsf{B}) \wedge parent(\mathsf{C},\mathsf{A}) \wedge \neg gen\_carrier(\mathsf{C}) \\ \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \wedge \neg gen\_carrier(\mathsf{A})$$

$$female(\mathsf{A}) \wedge parent(b,a) \wedge \neg gen\_carrier(\mathsf{B}) \wedge parent(\mathsf{C},\mathsf{A}) \wedge \neg gen\_carrier(\mathsf{C}) \\ \wedge \mathsf{A} \neq \mathsf{B} \wedge \mathsf{B} \neq \mathsf{C} \wedge \mathsf{A} \neq \mathsf{C} \wedge \neg color\_blind(\mathsf{A})$$

A.3 *RME formalization of* `colorblind`

$$(gen\_carrier(\mathsf{A}) \mid gen\_carrier(\mathsf{B}), \neg female(\mathsf{A}), color\_blind(\mathsf{B}), \\ parent(\mathsf{B},\mathsf{A}), A \neq B)[1.0]$$

$$(color\_blind(\mathsf{A}) \mid gen\_carrier(\mathsf{B}), \neg female(\mathsf{A}), color\_blind(\mathsf{B}), \\ parent(\mathsf{B},\mathsf{A}), A \neq B)[1.0]$$

$$(gen\_carrier(\mathsf{A}) \mid gen\_carrier(\mathsf{B}), \neg female(\mathsf{A}), \neg color\_blind(\mathsf{B}), \\ parent(\mathsf{B},\mathsf{A}), A \neq B)[0.5]$$

$$(color\_blind(\mathsf{A}) \mid gen\_carrier(\mathsf{B}), \neg female(\mathsf{A}), \neg color\_blind(\mathsf{B}), \\ parent(\mathsf{B},\mathsf{A}), A \neq B)[0.5]$$

$$(gen\_carrier(\mathsf{A}) \mid \neg female(\mathsf{A}), \neg gen\_carrier(\mathsf{B}), \\ parent(\mathsf{B},\mathsf{A}), A \neq B)[0.0]$$

$$(color\_blind(\mathsf{A}) \mid \neg female(\mathsf{A}), \neg gen\_carrier(\mathsf{B}), \\ parent(\mathsf{B},\mathsf{A}), A \neq B)[0.0]$$

$$(gen\_carrier(\mathsf{A}) \mid gen\_carrier(\mathsf{B}), female(\mathsf{A}), parent(\mathsf{B{,}A}), gen\_carrier(\mathsf{C}), \\ parent(\mathsf{CA}), color\_blind(\mathsf{C}), B \neq C,\ A \neq C,\ A \neq B)[1.0]$$

$$(color\_blind(\mathsf{A}) \mid gen\_carrier(\mathsf{B}), female(\mathsf{A}), parent(\mathsf{B},\mathsf{A}), gen\_carrier(\mathsf{C}), \\ parent(\mathsf{CA}), color\_blind(\mathsf{C}), B \neq C,\ A \neq C,\ A \neq B)[1.0]$$

$$(gen\_carrier(\mathsf{A}) \mid gen\_carrier(\mathsf{B}), female(\mathsf{A}), parent(\mathsf{B},\mathsf{A}), \neg color\_blind(\mathsf{C}), \\ gen\_carrier(\mathsf{C}), parent(\mathsf{CA}), B \neq C,\ A \neq C,\ A \neq B)[1.0]$$

$$(color\_blind(\mathsf{A}) \mid gen\_carrier(\mathsf{B}), female(\mathsf{A}), parent(\mathsf{B},\mathsf{A}), \neg color\_blind(\mathsf{C}), \\ gen\_carrier(\mathsf{C}), parent(\mathsf{CA}), B \neq C,\ A \neq C,\ A \neq B)[0.5]$$

$$(\mathit{gen\_carrier}(\mathsf{A}) \mid \mathit{gen\_carrier}(\mathsf{B}), \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \neg\mathit{color\_blind}(\mathsf{C}),$$
$$\neg\mathit{gen\_carrier}(\mathsf{C}), \mathit{parent}(\mathsf{CA}), B \neq C,\ A \neq C,\ A \neq B)[1.0]$$
$$(\mathit{color\_blind}(\mathsf{A}) \mid \mathit{gen\_carrier}(\mathsf{B}), \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \neg\mathit{color\_blind}(\mathsf{C}),$$
$$\neg\mathit{gen\_carrier}(\mathsf{C}), \mathit{parent}(\mathsf{CA}), B \neq C,\ A \neq C,\ A \neq B)[0.0]$$
$$(\mathit{gen\_carrier}(\mathsf{A}) \mid \neg\mathit{gen\_carrier}(\mathsf{B}), \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \mathit{gen\_carrier}(\mathsf{C}),$$
$$\mathit{parent}(\mathsf{CA}), \mathit{color\_blind}(\mathsf{C}), B \neq C,\ A \neq C,\ A \neq B)[1.0]$$
$$(\mathit{color\_blind}(\mathsf{A}) \mid \neg\mathit{gen\_carrier}(\mathsf{B}), \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \mathit{gen\_carrier}(\mathsf{C}),$$
$$\mathit{parent}(\mathsf{CA}), \mathit{color\_blind}(\mathsf{C}), B \neq C,\ A \neq C,\ A \neq B)[0.0]$$
$$(\mathit{gen\_carrier}(\mathsf{A}) \mid \neg\mathit{gen\_carrier}(\mathsf{B}), \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \neg\mathit{color\_blind}(\mathsf{C}),$$
$$\mathit{gen\_carrier}(\mathsf{C}), \mathit{parent}(\mathsf{CA}), B \neq C,\ A \neq C,\ A \neq B)[0.5]$$
$$(\mathit{color\_blind}(\mathsf{A}) \mid \neg\mathit{gen\_carrier}(\mathsf{B}), \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \neg\mathit{color\_blind}(\mathsf{C}),$$
$$\mathit{gen\_carrier}(\mathsf{C}), \mathit{parent}(\mathsf{CA}), B \neq C,\ A \neq C,\ A \neq B)[0.0]$$
$$(\mathit{gen\_carrier}(\mathsf{A}) \mid \neg\mathit{gen\_carrier}(\mathsf{B}), \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \neg\mathit{gen\_carrier}(\mathsf{C}),$$
$$\mathit{parent}(\mathsf{CA}), B \neq C,\ A \neq C,\ A \neq B)[0.0]$$
$$(\mathit{color\_blind}(\mathsf{A}) \mid \neg\mathit{gen\_carrier}(\mathsf{B}), \mathit{female}(\mathsf{A}), \mathit{parent}(\mathsf{B},\mathsf{A}), \neg\mathit{gen\_carrier}(\mathsf{C}),$$
$$\mathit{parent}(\mathsf{CA}), B \neq C,\ A \neq C,\ A \neq B)[0.0]$$

A.4 BLP *formalization of* `cold`

$$(\mathit{cold}(\mathsf{X}))$$
$$(\mathit{cold}(\mathsf{X}) \mid \mathit{susceptible}(\mathsf{X}))$$
$$(\mathit{cold}(\mathsf{X}) \mid \mathit{contact}(\mathsf{X},\mathsf{Y}), \mathit{cold}(\mathsf{Y}))$$

A.5 MLN *formalization of* `cold`

-0.5108	$\mathit{contact}(\mathsf{X},\mathsf{Y}) \wedge \mathit{cold}(\mathsf{Y}) \wedge \mathsf{X} \neq \mathsf{Y} \Rightarrow \mathit{cold}(\mathsf{X})$
-2.3026	$\mathit{susceptible}(\mathsf{X}) \Rightarrow \mathit{cold}(\mathsf{X})$
-2.9957	$\mathit{cold}(\mathsf{X})$
	$\neg\mathit{contact}(\mathsf{X},\mathsf{X})$
	$\mathit{contact}(\mathsf{X},\mathsf{Y}) \Leftrightarrow \mathit{contact}(\mathsf{Y},\mathsf{X})$

A.6 RME *formalization of* `cold`

$$(\mathit{cold}(\mathsf{X}) \mid \mathit{contact}(\mathsf{X},\mathsf{Y}), \mathit{cold}(\mathsf{Y}), \mathsf{X} \neq \mathsf{Y})[0.6]$$
$$(\mathit{cold}(\mathsf{X}) \mid \mathit{susceptible}(\mathsf{X}))[0.1]$$
$$(\mathit{cold}(\mathsf{X}))[0.05]$$
$$(\mathit{contact}(\mathsf{X},\mathsf{Y}) \mid \mathit{contact}(\mathsf{Y},\mathsf{X}), \mathsf{X} \neq \mathsf{Y})[1.0]$$
$$(\mathit{contact}(\mathsf{X},\mathsf{X}))[0.0]$$

A.7 BLP *formalization of* `randomwalk`

$$(\mathit{isLeft}(\mathsf{X}) \mid \mathit{previous}(\mathsf{X},\mathsf{Y}), \mathit{isLeft}(\mathsf{Y}))$$

A.8 MLN *formalization of* `randomwalk`

$$\begin{array}{ll} -0.3219 & \mathit{previous}(\mathsf{X},\mathsf{Y}) \wedge \neg \mathit{isLeft}(\mathsf{Y}) \Rightarrow \mathit{isLeft}(\mathsf{X}) \\ -2.3219 & \mathit{previous}(\mathsf{X},\mathsf{Y}) \wedge \mathit{isLeft}(\mathsf{Y}) \Rightarrow \mathit{isLeft}(\mathsf{X}) \end{array}$$

A.9 RME *formalization of* `randomwalk`

$$(\mathit{isLeft}(\mathsf{T1}) \mid \neg \mathit{isLeft}(\mathsf{T2}), \mathit{previous}(\mathsf{T1},\mathsf{T2}), \mathsf{T1} \neq \mathsf{T2})[0.8]$$
$$(\mathit{isLeft}(\mathsf{T1}) \mid \mathit{isLeft}(\mathsf{T2}), \mathit{previous}(\mathsf{T1},\mathsf{T2}), \mathsf{T1} \neq \mathsf{T2})[0.2]$$