The ICLP 2014 Doctoral Consortium

MARTIN GEBSER
Helsinki Institute for Information Technology HIIT
Department of Information and Computer Science
Aalto University, Finland

JAEL KRIENER
Microsoft Research–Inria
Joint Centre
Saclay, France

submitted [n/a]; revised [n/a]; accepted [n/a]

The ICLP 2014 Doctoral Consortium (DC) is the tenth doctoral consortium collocated with the 30th International Conference on Logic Programming (ICLP), the Federated Logic Conference (FLoC), and the Vienna Summer of Logic (VSL). The ICLP 2014 DC follows the very positive experience of previous events held in Sitges (Spain) on October 3rd, 2005, in Seattle (WA, USA) on August 21st, 2006, in Porto (Portugal) on September 8th, 2007, in Udine (Italy) on December 10th, 2008, in Pasadena (CA, USA) on July 15th, 2009, in Edinburgh (Scotland) on July 20th, 2010, in Lexington (KY, USA) on July 6th, 2011, in Budapest (Hungary) on September 4th, 2012, and in Istanbul (Turkey) on August 24th, 2013.

The ICLP 2014 DC, held in Vienna (Austria) on July 20th, 2014, provides a forum for doctoral students working in areas related to logic programming. We received eight applications, pointing out the high value and interest of such an event for young academics. It is a particular pleasure for us to acknowledge the eligibility and high quality of all applications, each of which was peer-reviewed by three program committee members or additional reviewers. The final ICLP 2014 DC program thus features eight contributions by doctoral students with the following countries of affiliation: Australia, Belgium, Egypt, Germany (two contributions), Italy, Poland, and United Kingdom. The proceedings of the ICLP 2014 DC provide the research summaries contributed by the student participants. We invite the reader to dwell into exciting research fields and topics explored by young academics. Please enjoy!

We thank the general chair, Manuel Carro, the program chairs, Michael Leuschel and Tom Schrijvers, as well as the workshop chair, Haifeng Guo, of the 30th International Conference on Logic Programming for their help in organizing the ICLP 2014 DC. We especially acknowledge the Association for Logic Programming (ALP) for financial travel support enabling the participation of doctoral students in the ICLP 2014 DC. We are grateful to the program committee as well as the additional reviewers for their excellent work to carefully review applications and to provide constructive feedback. Last but not least, we thank the student participants for their invaluable contributions, which are the key ingredient for the success of the ICLP 2014 DC!

July 2014,
Martin Gebser and Jael Kriener
Contributors

Rehan Abdul Aziz  Bound Founded Answer Set Programming
Joachim Jansen  Model Revision Inference for Extensions of First Order Logic
Grzegorz Jaśkiewicz  Logic Programming as Scripting Language for Bots in Computer Games – Research Overview
Michael Morak  The Impact of Disjunction on Reasoning under Existential Rules: Research Summary
Aya Saad  CDF-Intervals: A Reliable Framework to Reason about Data with Uncertainty
Nada Sharaf  Visualization of Constraint Handling Rules
Heike Stephan  Application of Methods for Syntax Analysis of Context-Free Languages to Query Evaluation of Logic Programs
Riccardo Zese  Reasoning with Probabilistic Logics

Officials

Program Chairs

Martin Gebser  Aalto University
Jael Kriener  Microsoft Research–Inria

Program Committee

Fabio Fioravanti  University of Chieti-Pescara
Ekaterina Komendantskaya  University of Dundee
Alessandra Mileo  National University of Ireland
Jose F. Morales  IMDEA Software Research Institute
Francesco Ricca  University of Calabria
Mehrnoosh Sadrzadeh  Queen Mary University of London
Takehide Soh  University of Kobe
Frank D. Valencia  École Polytechnique de Paris
Joost Vennekens  K.U. Leuven
Antoine Zimmermann  École Nationale Supérieure des Mines de Saint-Étienne

Additional Reviewers

Matthias Nickles  National University of Ireland
Salim Perchy  École Polytechnique de Paris
Bound Founded Answer Set Programming

Rehan Abdul Aziz

Supervisors: Peter Stuckey and Geoffrey Chu
National ICT Australia, Victoria Laboratory,∗
Department of Computing and Information Systems
Room 6.22, Doug McDonell Building (Building 168),
The University of Melbourne, Australia
Email: raziz@student.unimelb.edu.au

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Answer Set Programming (ASP) is a powerful modelling formalism that is very efficient in solving combinatorial problems. ASP solvers implement the stable model semantics that eliminates circular derivations between Boolean variables from the solutions of a logic program. Due to this, ASP solvers are better suited than propositional satisfiability (SAT) and Constraint Programming (CP) solvers to solve a certain class of problems whose specification includes inductive definitions such as reachability in a graph. On the other hand, ASP solvers suffer from the grounding bottleneck that occurs due to their inability to model finite domain variables. Furthermore, the existing stable model semantics are not sufficient to disallow circular reasoning on the bounds of numeric variables. An example where this is required is in modelling shortest paths between nodes in a graph. Just as reachability can be encoded as an inductive definition with one or more base cases and recursive rules, shortest paths between nodes can also be modelled with similar base cases and recursive rules for their upper bounds. This deficiency of stable model semantics introduces another type of grounding bottleneck in ASP systems that cannot be removed by naively merging ASP with CP solvers, but requires a theoretical extension of the semantics from Booleans and normal rules to bounds over numeric variables and more general rules. In this work, we propose Bound Founded Answer Set Programming (BFASP) that resolves this issue and consequently, removes all types of grounding bottleneck inherent in ASP systems.

1 Motivation

Answer Set Programming (Baral 2003) is a useful modelling paradigm to solve search and planning problems. Modern ASP solving (Gebser et al. 2007; Gebser et al. 2012) builds on propositional satisfiability (SAT) solving (Mitchell 2005). However, ASP solvers have a competitive edge over SAT solvers in problems whose model involves some notion of transitive closure, e.g., reachability or connectivity in a graph. This is due to the difference in semantics of both systems; ASP solvers implement stable model semantics (Gelfond and Lifschitz 1988) which minimizes the number of variables that are true in a given logic program while a SAT solver only looks for an assignment that satisfies all the given clauses. In ASP, in order for a variable to be true, it must have some rule as a support that justifies it being true. Furthermore, no set of variables can circularly support one another. E.g. given two rules $a \leftarrow b$ and $b \leftarrow a$, the only valid solution in

∗ NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
stable model semantics is where \( a \) and \( b \) are both false, whereas in propositional semantics, both the variables being true is also a valid solution.

As ASP systems such as SMODELS and CLASP only deal with Boolean variables, they are inefficient for solving problems that are naturally modelled with integers, especially if they have large domains. These combinatorial problems (e.g. scheduling) are ubiquitous in Computer Science, which makes ASP a poor choice to model and solve them. The most obvious way to model these in ASP is to represent each integer’s domain as a set of Boolean variables and impose constraints on these variables to ensure consistency. This incurs what is commonly known as the **grounding bottleneck** problem. Naturally, decomposing a large finite domain to Boolean variables blows up the problem size in the size of domains of integer variables. Constraint Programming solvers (Rossi et al. 2006) and Mixed Integer Programming solvers, on the other hand, are excellent candidates for these problems as they support numeric variables natively. Unfortunately, constraint solvers suffer from the same inefficiency as SAT solvers regarding problems like reachability that require inductive definitions. A hybrid system that has the best of both worlds, i.e., inductive rules for Boolean variables from ASP, and native support for integers and constraints over them from CP, addresses both the concerns. However, in this work, we propose that even this hybrid approach is not sufficient, and there exists a type of grounding bottleneck that is still not removed by combining the strengths of CP and ASP solvers. Let us illustrate this by first looking at a benchmark from ASP competitions, and then modifying its problem description.

Consider the Minimum Connected Dominating Set (MCDS) problem. A dominating set is a set of nodes such that every node in the graph is either in the set or has at least one neighbour in the set. The objective is to find a dominating set of minimum cardinality such that the subgraph induced by dominating nodes is connected. Let us look at the ASP encoding of the problem \(^1\).

A vertex \( X \) is given in the input as \( \text{vtx}(X) \), an edge from \( X \) to \( Y \) as \( \text{edge}(X,Y) \) and the edge relation is symmetric.

\[
\begin{align*}
R_1 & \quad \{ \text{dom}(U) : \text{vtx}(U) \}.
R_2 & \quad \text{in}(V) \leftarrow \text{dom}(V).
R_3 & \quad \text{in}(V) \leftarrow \text{edge}(U,V) \land \text{dom}(U).
C_1 & \quad \text{vtx}(U) \land \neg \text{in}(U).
R_4 & \quad \text{reach}(U) \leftarrow \text{dom}(U) \land \exists V < U : \text{dom}(V).
R_5 & \quad \text{reach}(V) \leftarrow \text{reach}(U) \land \text{dom}(V) \land \text{edge}(U,V).
C_2 & \quad \text{dom}(U) \land \neg \text{reach}(U).
O & \quad \text{minimize}\{\text{dom}(U) : \text{vtx}(U)\}.
\end{align*}
\]

\( R_1 \) introduces a decision variable \( \text{dom} \) for each vertex specifying whether it is a dominating vertex or not. \( R_1 \) is a choice rule which means that for a given node \( U \), this rule can be a justification for \( \text{dom}(U) \) if it is true. \( R_2, R_3 \) and \( C_1 \) model that every node must either be a dominating node or have a neighbour that is dominating. This is done with the help of an auxiliary predicate \( \text{in} \) that becomes true when at least one of the conditions is met. \( C_1 \) says that there can be no node for which \( \text{in} \) is false. \( R_4 \) and \( R_5 \) define the predicate \( \text{reach} \) that is used to model the connectivity constraint of the induced subgraph by the dominating nodes. \( R_4 \) encodes the base case for reachability, specifying that the node with the lowest index is reachable by definition.

\(^1\) Based on the model from Potassco group in the second ASP competition: http://dtai.cs.kuleuven.be/events/ASP-competition/encodings.shtml
This choice is arbitrary and its purpose can be satisfied by any criterion to select a dominating node. $R_5$ is a recursive case for reachability and it says that a dominating neighbour of a reachable node is also reachable. The constraint $C_2$ says that all dominating nodes must be reachable. The objective, given by $O$, is to minimize the cardinality of the dominating set.

Let us modify MCDS such that the edges in the graph also have weights (edge$(U, V, W)$ means that the edge from $U$ to $V$ has weight $W$) and there is an additional constraint that the diameter (maximum distance between any two nodes) of the dominating subgraph is less than a certain given value $K$. This problem has applications in computer networks (Kim et al. 2009; Buchanan et al. 2013). Let $d(X, Y)$ represent the distance (shortest path) between two dominating nodes $X$ and $Y$. In MCDS, it is sufficient to check for reachability of every dominating node from an arbitrary node to ensure connectedness. However, to enforce the new constraint, we need a distance variable for each pair of nodes in the dominating set. We can replace $R_4$, $R_5$ and $C_2$ in the above encoding of MCDS with the following:

$$R_4 \quad \text{d}(U, U) \leq 0 \leftarrow \text{dom}(U).$$

$$R_5 \quad \text{d}(U, T) \leq \text{d}(V, T) + W \leftarrow \text{dom}(T) \wedge \text{dom}(U) \wedge \text{dom}(V) \wedge \text{edge}(U, V, W).$$

$$C_2 \leftarrow \text{dom}(U) \wedge \text{dom}(V) \wedge \text{d}(U, V) > K.$$

$R_4$ is the base case for $d$ and it says that the distance from a dominating node to itself is at most 0. $R_5$ is a recursive rule that specifies that for two dominating neighbours and a dominating node $T$, the distance between one end of the node to $T$ is at most the distance between the other end and $T$, plus the weight of the edge. Finally, the constraint $C_2$ establishes that the distance between any two dominating nodes must be at most $K$. It is unnecessary to include the previous reachability rules since finite distances between all pairs of dominating nodes implies that the dominating set is connected.

Rules like $R_4$ and $R_5$ on integer variables are clearly not supported by current ASP systems. The semantics that we wish to associate with the distance variables is that firstly, if there are no rules supporting them, then they are equal to $\infty$. Secondly, any rule for a distance variable justifies a value lower than $\infty$ and thirdly, the upper bounds of these variables cannot form a circular justification. E.g. if there are two rules: $a \leq b$ and $b \leq a$, then any solution where $a$ and $b$ are equal to a finite value should be rejected, and the only stable solution should be one where both are equal to $\infty$. The distance variable is essentially an upper-bound founded (ub-founded) variable, for which the upper bound needs to be justified by some rule. We can encode these upper-bound founded distance variables in ASP along with our desired semantics by replacing $R_4$, $R_5$, and $C_2$ as follows:

$$d_{ub}(U, V, N) \leftarrow \text{dom}(U) \wedge \text{dom}(V) \wedge d_{ub}(U, V, N - 1), N < M.$$

$$d_{ub}(U, U, 0) \leftarrow \text{dom}(U).$$

$$d_{ub}(U, T, D + W) \leftarrow \text{dom}(T) \wedge \text{dom}(U) \wedge \text{dom}(V) \wedge \text{edge}(U, V, W) \wedge \text{d}(V, T, D).$$

$$\text{d}(U, V, D) \leftarrow \text{dom}(U) \wedge \text{dom}(V) \wedge d_{ub}(U, V, D) \wedge \neg d_{ub}(U, V, D - 1).$$

$$\wedge \text{d}(U, V, D) \wedge D > K.$$

In the above encoding, $M$ is a sufficiently large integer and $d_{ub}(U, V, N)$ specifies that the distance between the dominating nodes $U$ and $V$ is at most $N$ (the subscript ub stands for upper-bound). $d(U, V, D)$ is defined as the minimum value for which $d_{ub}(U, V, D)$ is true. Unfortunately, an ASP solver on this encoding quickly runs into the grounding bottleneck problem as
we increase edge weights and the bound on diameter. This is the motivation of this work, i.e., to support founded numeric variables and rules like $R_4$ and $R_5$ without grounding them.

The symmetric analog for a ub-founded variable is a lower-bound founded (lb-founded) variable, which is by default equal to $-\infty$ (false for Boolean) and further rules for it can justify a greater value on its lower bound. In this generalization, ASP variables are simply lb-founded Boolean variables. For simplicity, we only consider lb-founded variables, and refer to them as founded variables. This simplification is possible because we can replace all ub-founded variables, their rules, and their constraints by corresponding lb-founded variables with similar rules and constraints. E.g., for the above problem, let $d(U,V)$ represent the negative of the distance between $U$ and $V$, then we can perform this transformation as follows:

$$R_4 \quad d(U,U) \geq 0 \leftarrow \text{dom}(U).$$

$$R_5 \quad d(U,T) \geq d(V,T) - W \leftarrow \text{dom}(T) \land \text{dom}(U) \land \text{dom}(V) \land \text{edge}(U,V,W).$$

$$C_2 \quad \leftarrow \text{dom}(U) \land \text{dom}(V) \land d(U,V) < -K.$$

An important point in the encoding of MCDS is that if we remove the reachability condition from the problem specification, and $R_4$, $R_5$, and $C_2$ from the encoding, then the problem can be solved by a SAT solver just as efficiently as an ASP solver. Recall that the only shortcoming of SAT solvers is related to modelling properties like reachability since the propositional semantics that they are based on does not naturally model recursive definitions. This leads us to the important observation that besides founded variables like $\text{reach}$ and $d$, there can be variables in a problem like the $\text{dom}$ variables that are not founded. Let us call them standard variables, owing to the fact that these are the usual variables in CP solvers. Standard variables can be assigned any value as long as all the constraints associated with them are satisfied. Founded variables, on the other hand, need rules to define their values and without them, they are equal to some default value.

Since there are no rules and founded variables in CP and MIP solvers, MCDS with bounded diameter as defined above cannot be efficiently solved by them. The MIP formulation as given in (Buchanan et al. 2013) encodes each distance variable with $K$ propositional variables, meaning that the problem size increases with $K$. Our encoding above that uses founded variables does not suffer from this problem. This leads us to distinguish between the two types of grounding bottlenecks. One is caused in a system by the absence of its support for standard integer variables. Let us call this type the standard grounding bottleneck. The other type of grounding bottleneck is caused by the lack of a system’s capabilities to handle founded numeric variables, therefore, let us call it founded grounding bottleneck.

In the next section, we formally define the semantics of Bound Founded Answer Set Programming (BFASP), a formalism that generalizes the stable model semantics to bounds over numeric variables, and allows for a richer set of rules for founded variables.  

2 Bound Founded Answer Set Programming

Let $\mathcal{V}$ be the set of variables. We consider three types of variables: integer, real, and Boolean. Furthermore, we divide the set of variables in two disjoint sets: standard $\mathcal{S}$ and founded variables $\mathcal{F}$. A domain $D$ maps each variable $x \in \mathcal{V}$ to a set of constant values $D(x)$. A valuation (or

\footnote{The \texttt{MiniZinc} encoding of MCDS with bounded diameter in BFASP is given Appendix A.}
assignment) $\theta$ over variables $vars(\theta) \subseteq V$ maps each variable $x \in vars(\theta)$ to a value $\theta(x)$. A constraint $c$ is a set of assignments over the variables $vars(c)$, representing the solutions of the constraint. Given a constraint $c$, a variable $y \in vars(c)$ is monotonically increasing (decreasing) in $c$ if for all solutions $\theta$ that satisfy $c$, increasing (decreasing) the value of $y$ also creates a solution, that is $\theta'$ where $\theta'(y) > \theta(y)$, and $\theta'(x) = \theta(x)$, $x \in vars(c) - \{y\}$, is also a solution of $c$.

A positive-CP $P$ is a collection of constraints where each constraint is increasing in exactly one variable and decreasing in the rest. The minimal solution of a positive-CP is an assignment $\theta$ that satisfies $P$ s.t. there is no other assignment $\theta'$ that also satisfies $P$ and there exists a variable $v$ for which $\theta'(v) < \theta(v)$. Note that for Booleans, true $>$ false. A satisfiable positive-CP $P$ always has a unique minimal solution. If we have bounds consistent propagators for all the constraints in the program, then the unique minimal solution can be found simply by performing bounds propagation on all constraints until a fixed point is reached, and then setting all variables to their lowest values.

A rule $r$ is a pair $(c, y)$ where $c$ is a constraint, $y \in \mathcal{F}$ is the head of the rule and it is increasing in $c$. A bound founded answer set program (BFASP) $P$ is a tuple $(S, \mathcal{F}, C, R)$ where $C$ and $R$ are sets of constraints and rules respectively. Given a variable $y \in \mathcal{F}$, $rules(y)$ is the set of rules with $y$ as their heads.

The reduct of a BFASP $P$ with an assignment $\theta$, written $P^\theta$, is a positive-CP made from each rule $r = (c, y)$ by replacing in $c$ each variable $x \in vars(c) - \{y\}$, if it is a standard variable or if $c$ is not decreasing in it, by its value $\theta(x)$ to create a positive-CP constraint $c'$. Let $r^\theta$ denote this constraint. If $r^\theta$ is not a tautology, it is included in the reduct. An assignment $\theta$ is a stable model of $P$ iff i) it satisfies all the constraints in $P$ and ii) it is the minimal model that satisfies $P^\theta$.

Example 1
Consider a BFASP with standard variable $s$, integer founded variables $a, b$, Boolean founded variables $x$ and $y$, and the rules: $(a \geq 0, a), (b \geq 0, b), (a \geq b + s, a), (b \geq 8 \leftarrow x, b), (x \leftarrow \neg y \land (a \geq 5), x)$. Consider an assignment $\theta$ s.t. $\theta(x) = true, \theta(y) = false, \theta(b) = 8, \theta(s) = 9$ and $\theta(a) = 17$. The reduct of $\theta$ is the positive-CP: $a \geq b + 9, b \geq 8 \leftarrow x, x \leftarrow a \geq 5$. The minimal model that satisfies the reduct is equal to $\theta$, therefore, $\theta$ is a stable model of the program. Consider another assignment $\theta'$ where all values are the same as in $\theta$, but $\theta'(s) = 3$. Then, $P^\theta'$ is the positive-CP: $a \geq 0, b \geq 0, a \geq b + 3, b \geq 8 \leftarrow x, x \leftarrow a \geq 5$. The minimal solution that satisfies this positive-CP is $M$ where $M(a) = 3, M(b) = 0, M(x) = M(y) = false$. Therefore, $\theta'$ is not a stable model of the program.

3 Overview of the existing literature
There are several approaches in the literature that aim at removing the standard grounding bottleneck from ASP systems. A majority of these approaches work as follows: they introduce finite domain integer variables and constraints inside the ASP program. The ASP solver passes these to a CP solver while maintaining a Boolean variable to represent the truth value of each constraint that is in the ASP program. For a constraint $c$ that appears in the program, this is done by reifying the constraint, i.e., introducing a Boolean variable $b$ to represent whether the constraint is true. The ASP and CP solvers communicate using these introduced Boolean variables. E.g., if $b$ is set true by the ASP solver, then the constraint $c$ is enforced by the CP solver. Since the ASP solver treats CP as a blackbox, it cannot directly learn clauses from the propagation performed by
the CP solver. Examples of systems that use this approach are the AC SOLVER algorithm (Mel-larkod et al. 2008), CLINGCON (Gebser et al. 2009) and EZCSP (Balduccini 2009). Recently, some systems have been introduced that overcome the limited learning by using a single solver that supports both founded Booleans as well as standard integer variables and constraints over them. One way to achieve this is to introduce standard integer variables inside an ASP solver, and extending ASP’s propagation engine to work like a CP solver (de Cat et al. 2013; Drescher and Walsh 2012). The second approach is given in our earlier work (Aziz et al. 2013a), and extends an existing CP solver with founded Boolean variables and normal rules. To implement the stable model semantics over these, it implements the source pointer technique (Simons et al. 2002) to prune unfounded sets (Van Gelder et al. 1988) of variables as a propagator, similar to the ASP solver CLASP (Gebser et al. 2012).

Translating in terms of its supported features a specification that is missing in a system is another way to remove standard grounding bottleneck. There are two approaches in the literature to accomplish this. The first approach provides a translation from an ASP program augmented with numeric variables and constraints to a Mixed Integer Program (Liu et al. 2012). As discussed earlier, the non-recursive parts of the program are straight-forward to translate. The non-trivial part is encoding rules that involve positive recursion. This is done using the level ranking mapping as given in (Janhunen 2004). The fundamental idea of the translation is that if there is an unfounded set in the solution of the original program, then the mapping contains an inconsistent set of inequalities. The second approach (Drescher and Walsh 2010) encodes entire CP solving into ASP using the well-known eager CP decompositions to SAT. Unfortunately, this a priori translation of CP to SAT is already known to be highly inefficient in the CP community where it is much more efficient to translate lazily as in lazy clause generation (Ohrimenko et al. 2009).

Compared to the standard grounding bottleneck, the focus on the founded grounding bottleneck has been relatively weak. The formalism that is closest to BFASP in terms of removing this bottleneck is Fuzzy Answer Set Programming (FASP) (Nieuwenborgh et al. 2006; Blondeel et al. 2013). The fuzzy atoms in FASP correspond to founded real variables in BFASP, and each logical connective in FASP can be expressed as a rule form in BFASP. We have provided the translation in our previous work (Aziz et al. 2013), showing that BFASP subsumes FASP. Most importantly, from the implementation point of view, the MIP based unfounded set detection algorithm (Janssen et al. 2008) given for FASP only detects unfounded sets in a complete solution, which means that it cannot prune partial solutions that contain unfounded sets. Therefore, the algorithm has a similar shortcoming as CMODELS (Lierler and Maratea 2004) in case of Boolean unfounded sets. Finally, lack of any good implementation for FASP makes it infeasible to carry out an empirical comparison of BFASP and FASP.

4 Goals and current status of the research
The broader goal of my PhD is to analyze the strengths and implementation techniques of ASP in order to enhance the existing modelling and solving capabilities of constraint solvers. This overall goal can be divided into the following subgoals.

- To define, study, implement, and evaluate BFASP in order to put it forward as a formalism, with an accompanying implementation, that does not suffer from any kind of grounding bottleneck. This subgoal has been completed, and the most important features of BFASP were published in ICLP 2013 (Aziz et al. 2013). The paper defines the semantics of BFASP.
and presents an unfounded set algorithm that detects circular sets of bounds and prunes them during search. It presents performance comparison of our implementation of BFASP with ASP on three benchmarks, and with CP on one benchmark, and the results demonstrate the need for BFASP.

Prior to introducing BFASP, we extended a CP solver with founded Boolean variables and normal rules (Aziz et al. 2013a). We implemented two known algorithms for unfounded set detection from the ASP literature inside the CP solver CHUFFED and compared it with CLINGCON on problems that involve inductive definitions as well as standard integer variables.

• To define and study the language of BFASP. As compared to ASP languages like GRINGO that follow a very restrictive grammar, the grammar for BFASP is very permissive and a user can write complex expressions as rules. Therefore, the first task in this subgoal is to simplify these rules to a small set of primitive rules. In other words, we want to extend the flattening principles (Stuckey and Tack 2013) used in constraint languages to BFASP. Secondly, ASP grounders use bottom-up grounding that generates as few useless rules as possible. These are rules that can be removed without affecting the stable solutions of the program. The second task is to generalize ASP bottom-up grounding technique for BFASP. Finally, magic set rewriting is a useful technique in logic programming that only instantiates rules that are relevant to a given query. Considering variables appearing in constraints and objective function comprise our query, i.e. the set of variables whose values are of interest, the final task is to generalize the magic set transformation for BFASP.

• Identify research areas and benchmarks where BFASP can be applied and that cannot be efficiently solved by the current ASP, CP, and Constraint ASP (Gebser et al. 2009) systems. The doctoral programme could prove especially beneficial with regard to this subgoal.

References


Appendix A  MINIZINC Encoding of Minimum Connected Dominating Set with Bounded Diameter

int: N; %number of nodes
int: E; %number of edges
array[1..E] of 1..N: from; %encodes an edge i (from[i], to[i])
array[1..E] of 1..N: to;
array[1..E] of int: weight; %weight of an edge
int: K; %bound on diameter
array[1..N] of var bool: dom; %whether a node is dominating

% Dominating set constraint
constraint forall (n in 1..N) (
    dom[n] \/~ exists(e in 1..E where from[e] = n) (dom[to[e]])
);

% Rules for negative distance
array[1..N,1..N] of var int: d :: founded;
rule forall (n in 1..N) (d[n,n] >= 0 :: head(d[n,n]));
rule forall (e in 1..E, n in 1..N) (
    d[from[e],n] >= d[to[e],n] - weight[e]
    <-> dom[from[e]] \/~ dom[to[e]] \/~ dom[n] :: head(d[from[e],n])
);

% Diameter constraint
constraint forall (u,v in 1..N where u != v) (
);

% Objective to minimize the cardinality of dominating set
solve minimize sum (n in 1..N) (bool2int(dom[n]));

% A toy instance
N=4;
E=6;
K=35;
from  =[1, 2, 2, 3, 3, 4];
to    =[2, 1, 3, 2, 4, 3];
weight =[20,20,30,30,40,40];
Model revision inference for extensions of first order logic

Joachim Jansen

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

I am Joachim Jansen and this is my research summary, part of my application to the Doctoral Consortium at ICLP’14. I am a PhD student in the Knowledge Representation and Reasoning (KRR) research group, a subgroup of the Declarative Languages and Artificial Intelligence (DTAI) group at the department of Computer Science at KU Leuven. I started my PhD in September 2012. My promotor is prof. dr. ir. Gerda Janssens and my co-promotor is prof. dr. Marc Denecker.

I can be contacted at joachim.jansen@cs.kuleuven.be or at:

Room 01.167
Celestijnenlaan 200A
3001 Heverlee
Belgium

An extended abstract / full version of a paper accepted to be presented at the Doctoral Consortium of the 30th International Conference on Logic Programming (ICLP 2014), July 19-22, Vienna, Austria

1 Problem description

1.1 Introduction

The IDP system is a state-of-the-art system for declarative problem solving; complex search-and optimization problems are solved in an efficient and generic manner. As time passes on however, the found solution has to be revised: new information (e.g., changed circumstances) has to be taken into account. In this case it is desirable to start from the old (near-)solution and by performing a limited amount of changes transform it into a solution in which the new information is processed. At the moment there are no efficient, general solutions for these kind of problems; the only way this problem is currently solved is by writing special-purpose algorithms. During my thesis I would like to devise a general way to solve these problems using IDP as a system supporting an expressive modeling language.

The Knowledge Base System (KBS) (De Pooter et al. 2011) paradigm is a declarative approach in which one specifies what needs to be solved, instead of writing procedures that depict how to do this (Apt 2003; Gebser et al. 2012). A KBS represents the knowledge in its explicit form using an expressive modeling language and provides inferences to solve different kinds of problems. The expressive modeling language has as advantage that domains with a very complex or quickly changing knowledge can be expressed in a concise and clear way. Additionally, knowledge can be reused to solve different problems sharing the same scope. Because the inferences are domain independent, they can be reused across different scopes as well.

One of these inferences is model revision; the adaptation of an existing solution to new
information. In a train dispatching toy-problem for example there are a plethora of unforeseen circumstances (e.g., track defects, delays, copper cable thefts) and the dispatching schedule needs to be adapted to new requirements. Model revision also tries to maintain as much as possible of the original solution (dispatching schedule) when processing the change. This is a consequence of the solution technique that is generally efficient (start from the ‘old’ solution and apply a limited amount of changes), but is also a desirable property of the computed solution. Indeed, when a train is delayed in Paris, it doesn’t make a lot of sense to change the dispatching schedule of trains in London when this is not necessary.

1.2 Model revision: a motivating example

Here we introduce a small motivating example of the model revision inference using the situation depicted in Figure 1.2. In this figure the train tracks are indicated using grey lines between nodes. The train starting in S1 (which we will also call Train1) has to go past stations Brussels and London and the train that starts in S2 ((which we will call Train2) has to visit stations Brussels and Paris. The dispatched route for this is indicated using a green dotted line, that of Train2 is indicated with a red dotted line. Imagine the train track between S1 (Shunting 1, shunttings are intermediary crossroads in train tracks where one can change direction) and P1B (Platform 1 in Brussels) is detected to have broken down. By using model revision we can construct a new route for Train2 in S1 that does not use any broken down train tracks. Figure 2(a) shows a high-quality revised model: a route has been found for Train1 without changing too much to the existing dispatching. Figure 2(b) shows a low-quality revised model: the route for Train1 is correct but an unnecessary change to the route of Train2 was made. The change to the route of Train2 was needed because there is a requirement that states that two trains cannot enter the same station on the same platform (at the same time).

1.3 Formal definition of model revision

The formal definition for the model revision problem is as follows (Witocx et al. 2009):

Given a FO(·) theory $T$, a model $M$ for this theory and a collection of domain atoms $C$. Henceforth $C$ are called the required changes. In the example $C$ is the usage of the tracks between S1 and P1B. Solving model revision for $(T, M, C)$ means searching a new model $M'$ of $T$ such that all domain atoms in $C$ all have a different value compared to their old one in $M$. $M'$ is also called the revised model. Figure 2 shows two possible revised models for the example; the broken down train tracks are not used in either case.
In addition to the required changes, one usually has to change other parts of the original solution as well to construct the revised model. We call these other changes between $M$ and $M'$ the additional changes and denote them with $S$. In the example the usage of the new route is the additional change.

Often it is not desirable that the entire original model $M$ is be changed; some elements are immutable. In the example the structure of the train tracks is considered immutable: we are not interested in new solutions that would require us to build additional train tracks (e.g., one between London and Paris). These immutable elements in the problem domain are represented by the limitation $G$, a set of domain atoms whose value must remain fixed. The revision problem for $(T, M, C, G)$ is the same as the revision problem for $(T, M, C)$, except for the extra condition that the additional changes cannot include any of the limitations (i.e., $S$ is disjoint with $G$).

2 Existing Literature

Model revision allows us to flexibly use with a computed solution by imposing new restrictions. Although this kind of flexible reasoning is essential to a KBS, there is no research for model revision (in its general sense) in the context of an expressive modeling language. Comparable research has been performed in areas of incremental constraint programming (Freeman-Benson et al. 1990) and reactive answer set programming (Gebser et al. 2011). In this research only a limited form of new requirements are supported: one takes into account specific forms of new types of knowledge, but e.g. there is no way to apply previously unforeseen changes. Recent research are also trying to tackle this problem on the SAT level (Abo et al. 2011). These SAT-level techniques are interesting for the implementation that will be provided eventually because IDP uses a SAT solver in its workflow, but do not work in the context of a complex modeling language. There has also been work on trying to construct the solution in such a way that it is ‘robust’ w.r.t. changes (Climent et al. 2014). For first order logic there is a basic algorithm that takes general changes into account (Wittock et al. 2009). This will serve as a starting point for my thesis.

3 Background

This section contains a short introduction to the used terminology. The following concepts are introduced briefly: Knowledge Base System paradigm (De Pooter et al. 2011), FO(·) (Blockeel et al. 2013), and the IDP system (De Cat et al. 2013; IDP 2013).
Each declarative system requires a language in which the problems are represented. This language is preferably expressive, so the problem domain can be intuitively expressed. The $\text{FO(·)}$ family of languages has been developed at the KRR group for this purpose.

$\text{FO(·)}$ is a family of expressive knowledge representation languages that extend classical First Order Logic (FO) with various concepts. Apart from the logical symbols ($\land$, $\lor$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, $\exists$, $\forall$), $\text{FO(·)}$ also contains:

- **Inductive definitions** are represented as a set of defining rules.
- **Set expressions** of the form $\{ x \ y : p(x) \land q(y) \land r(x, y) \}$ represent the set of all combinations of $x$ and $y$ such that $p(x) \land q(y) \land r(x, y)$.
- **Aggregates** express the result of an aggregate function of a set expression together with a cost function (for each element in the set). The following aggregate functions are supported: minimum, maximum, sum, product and cardinality.
- **Expressive quantifiers** such as $\exists = 1$ (there exists exactly one), $\exists \geq 2$ (there exists at least two) ...
- **Types and subtypes** : each variable is typed in $\text{FO(·)}$.
- **(Partial) functions** . These are non-Herbrand functions.
- **Arithmetic operators** such as $+, -, \times, \div, |x|, \%$.

A problem specification in $\text{FO(·)}$ consists of at least three parts: a **vocabulary** that depicts the domain ontology, a **theory** containing the constraints for this problem, and a **structure** that contains the known data about the problem.

For a more hands-on introduction to $\text{FO(·)}$ and IDP, the reader is directed to our webpage of examples at [http://dtai.cs.kuleuven.be/krr/software/idp-examples](http://dtai.cs.kuleuven.be/krr/software/idp-examples).

### 3.2 Knowledge Base System

In a Knowledge Base System (KBS) the data and knowledge (expressed in the modeling language, e.g., $\text{FO(·)}$) are maintained in a **Knowledge Base**. A KBS then offers a variety of inferences to solve problems with the knowledge. A conceptual representation of a KBS is displayed in Figure 3.

Among these inferences are **model expansion** (extend a three-valued structure such that it satisfies a theory), **model checking** (verify whether a given structure satisfies a theory), **optimization** (extend a three-valued structure to a two-valued structure that satisfies a theory that has the least cost), and **model revision** (see Section 1.3).
3.3 The IDP system

The IDP system is a state-of-the-art implementation of the KBS paradigm using FO(·)IDP as its modeling language. The workflow of the IDP system is as follows (De Cat et al. 2013). First the FO(·) theory is ground into a low-level propositional representation. This representation is called “Extended CNF” or ECNF. It is an extension of CNF with concepts such as inductive definitions (that are ground). Next IDP uses a SAT-solver, MINISAT(ID), to generate solutions based on the grounding. IDP as well as MINISAT(ID) are open-source and available at https://bitbucket.org/krr/idp and respectively https://bitbucket.org/krr/minisatid.

The goal of my thesis is to provide support for model revision in the IDP system.

4 Goal of the research

The goal of my PhD thesis is to develop logic inference methods for different forms of model revision in the context of the FO(·) modeling language.

In order for this to be possible, we need a mechanism to reason about changes propagating through a theory. To this end, the approximating definition for a theory (Wittocx 2010; Vlaeminck 2012; Vaezipoor et al. 2011) needs to be computed and used to propagate impact of a change to the solution throughout the theory of the problem. The theory behind this currently supports basic FO. For my thesis, I will extend the scope of approximating definition to theories containing more expressive constructs such as inductive definitions, aggregates...

Because the approximating definition is a definition that needs to be calculated, there need to be efficient techniques for doing so. It was proposed in (Wittocx 2010) that the definition can be evaluated using any external system that can evaluate definitions (or rules).

For model revision there are typically a multitude of possible revisions. There is a need for proper criteria that quantify the quality of a revision. I intend to construct criteria using a domain independent as well as a domain dependent approach. For the domain independent criteria some brute-force metrics such the number of changed domain atoms will be used. In order to properly support domain dependent criteria, a user needs to able to express which revisions are preferred over others. This can be done either by expressing them beforehand using some sort of cost function. For this the knowledge representation language needs to be extended. Another way to do this is to let the user interactively guide the search process for the revision, indicating which choices are preferred.

5 Current status of the research

The first part of my PhD consisted of constructing an interface between XSB and IDP for calculating definitions that can be completely evaluated. For this work, the inductive definitions are transformed into rules for tabled Prolog. This was published in TPLP (Jansen et al. 2013).

Further I extended IDP to compute the approximating definition using the existing theory concerning this topic. Additionally, IDP was also extended with the possibility to making the input structure as two-valued as possible before grounding using the approximating definition (De Cat et al. 2013) as an alternative approach to the “Ground With Bounds” (GWB) technique depicted in (Wittocx 2010; Vlaeminck 2012). Currently benchmarks are being run to compare the two approaches. According to (Vaezipoor et al. 2011) the new approach using approximating definition outperforms the classical GWB technique because it will always
compute all possible unit propagation possible (at SAT-level) beforehand. GWB on the other hand sometimes performs cutoffs to increase performance. Preliminary results however contradict this claim.

Another claim from (Vaezipoor et al. 2011) is currently being investigated: a “smarter” grounding will affect the search tree as well. A smarter grounding can contain fewer introduced symbols (i.e., Tseitins) because it was detected beforehand that they need not be generated at all. Since these Tseitins are not removed by performing unit propagation at SAT-level, a smarter grounding thus contains (according to the above authors) possibly fewer “autarkies” - irrelevant parts of the search space in which the solver possibly can waste time. Currently experiments are being run that compare the search behaviour of solver runs on smart, respectively “naive” groundings.

6 Preliminary results

Benchmarks over problems in the $P$ complexity class that are generally solved by evaluating definitions for completely given structures show that a great speedup is achieved compared to the classical approach (Jansen et al. 2013).

Preliminary results (a complete study is being performed) suggest that making the input structure as two-valued as possible before grounding using approximating definitions is not superior to its counterpart the classic GWB workflow already implemented in IDP. Additionally, there were only very few problems where the grounding was smaller.

7 Open Issues

Tasks that still need addressing are the extension of the approximating definition for theories that contain more expressive constructs such as inductive definitions, aggregates... Additionally, the solver MINISAT(ID) will need to be adapted to support model revision. For support of interactively searching for a revision, the solver workflow also needs to be updated to work interactively with user input.
References


Logic Programming as Scripting Language for Bots in Computer Games – Research Overview

Grzegorz Jaśkiewicz
Warsaw University of Technology, Poland
(e-mail: grzegorz@jaskiewi.cz)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract
This publication is to present a summary of research (referred as κ-Labs1) carried out in author’s Ph.D studies on topic of application of Logic Programming as scripting language for virtual character behavior control in First Person Shooter (FPS) games.

The research goal is to apply reasoning and knowledge representation techniques to create character behavior, which results in increased players’ engagement.

An extended abstract / full version of a paper accepted to be presented at the Doctoral Consortium of the 30th International Conference on Logic Programming (ICLP 2014), July 19-22, Vienna, Austria

KEYWORDS: Logic Programming, Video Games, Virtual Characters, Scripting Language, Decision Rules

Contents

1 Introduction
2 Related works
  2.1 Rule-based systems in video games development
  2.2 Frameworks for developing rational agents
3 Research Overview
  3.1 Research status summary
  3.2 Bot architecture elaborated during research
4 Experiments and Results
  4.1 Offline experiments
  4.2 Online experiments
  4.3 Performance
5 Future Works
References

1 website available at http://www.kappalabs.org
1 Introduction

The First Person Shooter (FPS) is a type of video game where a player sees the level from the eyes of the character being played. Counter-Strike is one of the most popular and successful FPS games in the world. It is one of the top 10 online games on the Steam network with a daily peak of online players near 40,000\(^2\) and it is played professionally around the world in e-sport leagues (Jana et al. 2007). Sample screenshots of the gameplay is presented in the fig. 1.

![First person perspective view in Counter-Strike game.](image)

The game is about fight of terrorist (Ts) and counter-terrorist (CTs) forces. Players are divided into those two teams and try to fulfill objectives, which are dependent on a map type. Most commonly played map types are defusion (DE) and counter-strike (CS). On CS maps CTs has to rescue hostages, which are guarded by Ts. Example of such map is cs_assau1t, where hostages are held in a warehouse, or cs_747, where hostages are held in a Boeing plane on an airfield. In DE maps CTs has to prevent Ts from bombing some important location.

The game was created in 1994 and its first versions were entirely multiplayer\(^3\). The lack of ability to play in single player mode resulted in community-developed Artificial Intelligence (AI) algorithms for replacing human controlled opponents. The software which controls characters in a multiplayer game is usually called “bot”.

Counter-Strike has large community of people who create modifications for the game (Kücklich 2005). As a result there are available many different bot implementations. Many of them are distributed as an open-source software. The research described in this article is based on the open-source bot implementation called E[POD]. The bot’s source code was altered to allow scripting the bot behavior using Logic Programming. The name \(\kappa\)Bot is used to refer to the bot which is an object of the research.

The idea of embedding a scripting language into video game is well-known practice in the game development industry. It allows to clearly divide responsibilities in project of creating a video game – game engine programmers and game world designers. In some cases game engine is not developed. Instead, the license is bought for usage in the particular product. The ability to script some aspects of gameplay makes such engine more flexible. Commonly, scripts are used to describe player interactions with game world, but character behavior is also part of this domain, hence there is a perfectly valid use-case for existence scripting tool for their behavior.

---

\(^2\) source: daily statistics for Counter-Strike 1.6 provided by the Steam – online gaming platform, see [http://store.steampowered.com/stats/](http://store.steampowered.com/stats/), access date: 07.2013.

\(^3\) game mode where multiple players participate in same game session; often played on a network (LAN or Internet).
Lua, Python and Unreal Script dominate as scripting languages for game engines (Anderson 2011). All of them are multiple paradigm languages: imperative, object-oriented, functional, procedural. However, none of them does allow logic programming by its syntax. This is distinct paradigm, which relies strongly on notion of reasoning and knowledge representation – the main concepts of rational agents. Virtual characters in FPS games could be regarded as rational agents, so the scripting language for character behavior in \( \kappa\text{Bot} \) allows logic programming.

2 Related works

2.1 Rule-based systems in video games development

The similarities presented in this section are based on application of rule-based systems for expressing a character behavior rules in video games. The most prominent examples of such solutions are:

- Generic Robot Language (Horswill 2000) – a simple functional language to define decision rules, which are compiled to C++ code;
- Goal Oriented Action Planning (Orkin 2005) – a performance-oriented planning algorithm based on \( A^* \) algorithm. Uses pre- and post-conditions based on a predicate calculus for actions;
- Adaptive game AI with dynamic scripting (Spronck et al. 2006) – a machine learning algorithm used to optimize a selection of preprogrammed decision rules for fighting characters;
- Cognitive Modeling Language (Funge 1998) – a language integrated with planning using \( A^* \) algorithm. It allows specifying constraints through logical rules on solutions being searched;
- Avatar Definition Language (Anderson 2005) – an imperative language to script conditions for triggering state changes for an underlying finite state machine. The script represents a high-level logic of a bot functioning, while states are executing actions for achieving simple goals.

2.2 Frameworks for developing rational agents

Multiagent scientific field is well-developed. There exists many tools and methods, which facilitate software development in this paradigm.

The examples of platforms for general multiagent development are JADE (Bellifemine et al. 2005) and SOAR (Laird et al. 1987). Those could be possibly used for controlling virtual characters in a game environment (Laird 2001).

There is also a software platform dedicated for that purpose: Pogamut (Gemrot et al. 2009).

The examples of planning algorithms for rational agents which are based on knowledge representation and reasoning are:

- GOAL (Hindriks 2009) - agents derive their choice of action from their beliefs and goals,
- FLUX (Thielscher 2005) - programming framework for rational agents based on the fluent calculus (Thielscher 1998).

The variety of tools for developing rational agent is a motivation to apply some of them for characters in video games.
3 Research Overview

The goal of the research is to explore capabilities of logic programming as scripting language for controlling virtual characters in FPS video games. The reason for starting this research was a presence of logic programming in a multiagent programming domain.

3.1 Research status summary

We have chosen Prolog as scripting language for κBot, because it is a general-purpose, Turing-complete scripting language. This is important, because it gives programming flexibility, so that a programmer can construct arbitrary algorithms using Prolog script. Still, the κBot scripts can benefit from declarative syntax, which is property of Prolog.

We have chosen a SWI-Prolog (Wielemaker 2003) as Prolog interpreter, because of unrestricted license and ease of embedding into software written in C++. The interpreter is not only a logic module, which computes a model for given rules and facts in order to make a decision. It is capable of invoking functions from the bot’s code. We made such decision to enhance Prolog usage as programming language, not only as an inference method. Typically, some parts of the script define agent’s reasoning rules and some are part of framework for reasoning, i.e. we use some generic predicates to divide inference rules into different generality levels: game rules, map type rules and map-specific rules.

We have used source code of E[POD] bot to create κBot bot. The original version of E[POD] relies on Brooks architecture (Zubek 2001) – the logic is divided into two layers with different level of abstraction:

- low-level reasoning – for expressing simple actions like moving around the map, attacking enemy, planting bombs, etc. Low-level reasoning algorithm tailored for particular task (e.g. planting a bomb) is called high-level action.
- high-level reasoning – for building bot behaviors by selecting of high-level actions.

The practice of separating task into several layers of abstraction is common practice in multiagent programming. For game bots two layers of abstraction is reasonable amount. However, for more complicated agents, more layers could be defined, e.g. robots, which must control their servomotors, may need additional hardware abstraction layers (Minsky 1986).

The goal is to develop a scripting module for the high-level reasoning layer. In first versions of κBot, E[POD]’s high-level decision-making algorithm was mimicked entirely by Prolog script. This was achieved by rewriting in Prolog all the rules and conditions which trigger change of high-level actions. Afterwards, rules for controlling bot behavior were divided into packages with different level of generality, i.e. general rules for playing the game, rules for playing on specific map type and rules for playing on concrete instance of the map. In recent versions κBot bot has predefined set of tactics scripted per map. Those tactics describe how to exploit map-specific features for gaining tactical advantage. For some maps they are even simulated bot negotiation for selecting collective team tactics, e.g. bot can vote according to its own, individual preferences, which tactic should it commit to.
3.2 Bot architecture elaborated during research

In this section we will show a core concept of the bot’s architecture, which is scriptable in Prolog. Many specific details like bots’ communication or collaboration are not covered by this section. However, any of those details are based upon concepts presented in this chapter.

Obviously, to make any decisions for any character the bot must learn information about the game environment it operates within. Relevant information are feed into Prolog script by invoking native predicates. Those can retrieve and transfer information, which are managed by the game engine, into Prolog interpreter, e.g. information about visible enemies, hearing footsteps, amount of ammunition and money available etc. An example of the declaration of such predicate is shown in the listing 1.

Listing 1. An example of native predicate declaration.
```
// predicate bot_in_fov( ?botID, ?botID)
// checks if a character is in a field of view
// of another character. May have multiple goals.
PREDICATE_NONDET(bot_in_fov, 2) { /* ... */ }
```

Another source of information is a database of dynamic clauses, which is implemented with help of assert/retract predicates. Those information are used to implement the bot’s memory. In contrary to information provided by the native predicates, memorized information can be changed freely by Prolog script. The example of information kept in bot’s memory are, e.g.

- information about a map topology, which change infrequently, e.g. hiding spots, ambush points,
- information about past actions, e.g. bot should not try to buy weapons more then once a round,

High-level actions are started by invoking a native predicate in the script. The backtracking of a proof does not cancel once created action. In order to keep control over the script execution, the bot script should be constructed in such way that these action predicates will not be backtracked. This often leads to decision rules in following form:

Listing 2. An example of starting an action from script.
```
do_reasoning(BotID) :-
    should_do_action(BotID, Args_for_action),
    do_action(BotID, Args_for_action).
```

where do_action in listing 2 is a native predicate for starting some action. Term should_do_action is condition for this action; backtracking can occur while proving it. The conditional expression also provides arguments, which are passed to the action being invoked.

The high-level actions has two important features: motivations and continuations. A motivation is a logical condition expressed through Prolog terms. The action is executed as long as motivation is evaluated to truth value (YES) according to information, which the bot receives from the environment. Introducing the motivation improves control over the bot by Prolog scripts, because it is possible to interrupt execution of high-level action. This technique also improves the separation between bot’s C++ code and Prolog scripts, because script creator does not have to know detailed specification of any particular action.

An example of a motivation:
Listing 3. An example of motivation for action.

```prolog
action_kill(
   BotID, EnemyID,
   and(bot_alive(EnemyID), danger_low(BotID))).
```

The action in listing 3 will make bot with identifier BotID try killing the bot with identifier EnemyID as long as invocation

```prolog
call(and(bot_alive(EnemyID), not_in_danger(BotID))).
```

evaluates to YES. This condition expresses conjunction of two sub-subconditions: the character referenced by EnemyID is alive and the character referenced BotID is not in danger. At the moment of checking the motivation both variables are instantiated — BotID is owner of task and EnemyID is the target to eliminate.

Continuations are simple form of planning and defining complex behaviors consisting of sequence of several high-level actions. There could be assigned a continuation to any high-level action. A continuation is a term in Prolog script which is executed after high-level action is successfully completed. As a result of executing continuation there could be created a new action to be executed. The new action could have its own motivation and continuation. Example of continuation usage:

Listing 4. An example of continuation for action.

```prolog
action_goto(BotID, Wp, andThen(
   action_liberate_hostages(BotID))).
```

The action in listing 4 will create an action for the character with identifier BotID, which will cause him go to a waypoint with identifier Wp. After reaching waypoint the continuation is started by executing

```prolog
call(action_liberate_hostages(BotID)).
```

This will create new action for the character to liberate hostages.

The schematics of dependencies between system components has been presented in diagram 2.

The dependencies in diagram 2 are following:

1. high-level actions change environment (e.g. move an agent),
2. high-level actions read environment state,
3. script execution may create new high-level actions,
4. motivations may cease action execution; continuations may create new actions,
5. reasoning, motivations and continuations may read and change contents of dynamic database, which constitute inner state,
6. perception predicates serves as a proxy for learning facts about game environment.

4 Experiments and Results

In experiments $\kappa$Bot has been always compared to $E[POD]$, because $\kappa$Bot without map-specific rules and knowledge plays no different than $E[POD]$. This is caused by close translation of
Logic Programming as Scripting Language for Bots in Computer Games – Research Overview

Counter-Strike
Game Engine

\(\kappa\text{Bot}\)

Game environment

High-level
Actions

Prolog
interpreter

Deliberative
and Reflex
Reasoning

Motivations
and Continuations

Inner
State

Perception
Predicates

Fig. 2. Draft of bot architectural software design.

E[POD]'s behavioral rules, expressed by C++ code, into Prolog script. This property is particularly useful, because it allows to attribute all changes in player perception of gameplay to modifications in high-level reasoning — all high-level actions, like aiming and navigating in map, were the same for the two compared bots.

4.1 Offline experiments

Offline experiments has been carried out, while \(\kappa\text{Bot}\) has been in development stage. Those kind of experiments allows to determine, if the bot is functioning effectively, i.e. does it not crash, does not consume too much CPU and is it able to reach its goals. Those experiments were conducted by gathering match statistics with \(\kappa\text{Bot}\) playing against E[POD].

Those experiments were carried out on cs_assault map. We have tracked a number of times each team was victorious and a number of times when team fulfilled its objective. We have defined fulfilling an objective as:

- for CTs, to rescue hostages, without killing the opposing team,
- for Ts, to prevent CTs from rescuing hostages, without the opposing team.

In table 1 and table 2 we present the results of offline testing of the version of \(\kappa\text{Bot}\), which was used in online experiments (section 4.2).

It could be seen that \(\kappa\text{Bot}\) gets better scores than E[POD]. Study of goal-fulfilled team victories suggests also that \(\kappa\text{Bot}\) is more goal-oriented than E[POD].

4.2 Online experiments

The primary idea of online experiments was to gather feedback and metrics from people who casually play Counter-Strike. The metrics for players engagement were obtained through obser-
Table 1. *Gameplay statistics of total team victories – 10 matches average.*

<table>
<thead>
<tr>
<th>CTs AI</th>
<th>Ts AI</th>
<th>CTs wins</th>
<th>Ts wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>E[POD]</td>
<td>E[POD]</td>
<td>4.1</td>
<td>6.4</td>
</tr>
<tr>
<td>κBot</td>
<td>E[POD]</td>
<td>8.6</td>
<td>5.3</td>
</tr>
<tr>
<td>E[POD]</td>
<td>κBot</td>
<td>1.9</td>
<td>7.0</td>
</tr>
<tr>
<td>κBot</td>
<td>κBot</td>
<td>6.6</td>
<td>8.4</td>
</tr>
</tbody>
</table>

Table 2. *Gameplay statistics of goal-fulfilled team victories – 10 matches average.*

<table>
<thead>
<tr>
<th>CTs AI</th>
<th>Ts AI</th>
<th>CTs wins</th>
<th>Ts wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>E[POD]</td>
<td>E[POD]</td>
<td>0.4</td>
<td>1.2</td>
</tr>
<tr>
<td>κBot</td>
<td>E[POD]</td>
<td>3.6</td>
<td>0.5</td>
</tr>
<tr>
<td>E[POD]</td>
<td>κBot</td>
<td>0.2</td>
<td>2.5</td>
</tr>
<tr>
<td>κBot</td>
<td>κBot</td>
<td>1.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

The analysis of player session time shows player tend to play with κBot to the end of the match. When they play with E[POD] they tend to disconnect from server before end of match more often. This could probably, be explained by fact that player notice the difference between well-known behavior of E[POD] bot and new behavioral specializations of κBot.
4.3 Performance

\( \kappa \text{Bot} \) performance was measured in order to keep resource consumption within realistic constraints for practically usable software. The presence of \( \kappa \text{Bot} \) in the online experiments is example of a server-side deployment. It differs from client-side deployment in terms of hardware and game engine functioning. Server machine usually has more CPU power than laptop or workstation. The server does not need to render 3D game environment scene, it only has to provide symbolic information about game state to its clients. Because of those difference tests were run for both settings. In general, the architecture seems to be usable: it is possible to define useful behavior without much impact on CPU. In client-side deployment Prolog interpreter took \( \approx 1\% \) of wall time. In server-side deployment \( \kappa \text{Bot} \) caused 7% raise of CPU utilization when compared to \( E[POD] \). Note that those two metrics are not comparable to each other, but each of them is appropriate to its application.

5 Future Works

The \( \kappa \text{Bot} \) architecture is a baseline for further extensions. We have tested a form of specialization of bot behavior using very simplified default logic. Currently we are investigating obtaining behavioral rules by computational methods.

Interpreters for some kind of logics, e.g. Golog, are implemented in Prolog or output a script which is executed in Prolog interpreter. This gives opportunities to test those logics as reasoning mechanism for \( \kappa \text{Bot} \), without burden of integration with bot’s C++ code.

Acknowledgements

I would like to thank my mentor prof. Jarosław Arabas for advices provided while writing this article.

References


The Impact of Disjunction on Reasoning under Existential Rules: Research Summary

MICHAEL MORAK
University of Oxford, Department of Computer Science, OX1 3QD, United Kingdom
(e-mail: michael.morak@cs.ox.ac.uk)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract
Datalog± is a Datalog-based language family enhanced with existential quantification in rule heads, equalities and negative constraints. Query answering over databases with respect to a Datalog± theory is generally undecidable, however several syntactic restrictions have been proposed to remedy this fact. However, a useful and natural feature however is as of yet missing from Datalog±: The ability to express uncertain knowledge, or choices, using disjunction. It is the precise objective of the doctoral thesis herein discussed, to investigate the impact on the complexity of query answering, of adding disjunction to well-known decidable Datalog± fragments, namely guarded, sticky and weakly-acyclic Datalog± theories. For guarded theories with disjunction, we obtain a strong 2EXP lower bound in the combined complexity, even for very restricted formalisms like fixed sets of (disjunctive) inclusion dependencies. For sticky theories, the query answering problem becomes undecidable, even in the data complexity, and for weakly-acyclic query answering we see a reasonable and expected increase in complexity.

A full version of a paper accepted to be presented at the Doctoral Consortium of the 30th International Conference on Logic Programming (ICLP 2014), July 19-22, Vienna, Austria

KEYWORDS: Ontological Reasoning, Query Answering, Existential Rules, Logic, TGDs

1 Introduction and Problem Description
For the last thirty years, Datalog (see e.g., (Abiteboul et al. 1995)) has played an important role as a conceptual query language. Whilst not directly implemented in mainstream database management systems (DBMS), it did heavily influence the design of the SQL standard, which now also allows for recursive statements, as can be expressed in Datalog.

However in recent years it has become increasingly important to add ontological reasoning capabilities to the existing object-relational querying capabilities of traditional DBMS: A query is no longer just evaluated over the extensional relational database, but also over an ontological theory that, using rules and constraints, describes how to derive new (intensional) knowledge from the extensional data. By extending Datalog in such a way that existential quantification, the first-order logic constant false and equalities between variables are permitted in the rule heads, this behaviour can be expressed. Recently the Datalog± family of languages has been proposed in (Calì et al. 2011), that defines sensible restrictions on the structure of such an ontological theory. These restrictions are necessary as, depending on the structure of the ontological theory, an infinite amount of intensional knowledge might be derivable, rising the question of decidability of this type of reasoning. Also, as new values can be invented along the way, the domain can become infinite.
Despite these obstacles, commercial service providers have already started to integrate ontological reasoning engines into their database management systems (see e.g., \cite{Oracle Inc. 2011, Microsoft Corp. 2011}), as there are several applications where such capabilities are desirable, such as data exchange, ontological reasoning (e.g., reasoning under description logics, or in the semantic web) and web data extraction.

**Problem Statement.** Given the fact that ontological reasoning is gaining mainstream acceptance and the fact that, as for example Answer Set Programming has proven, rule-based languages are well suited for knowledge representation and reasoning tasks, it is natural to ask how to enrich the languages we currently have with new, useful constructs. The construct that we want to focus on here is disjunction. Until now, Datalog\textsuperscript{\pm} rules only allow us to express deterministic knowledge. But what about natural statements like “every person has a parent that is either male or female” or “every student is either an undergraduate or a graduate student”? Such statements are not captured by existing Datalog\textsuperscript{\pm} languages. Seeing that disjunctive knowledge is an important feature in other logical languages like Answer Set Programming or Description Logics that allows users to intuitively formulate problems by, e.g., applying a guess-and-check approach, enriching Datalog\textsuperscript{\pm} with disjunction is therefore a logical next step.

The objective of my doctoral studies is thus to introduce the language feature of disjunction to Datalog\textsuperscript{\pm}, and investigate in-depth what the impact of doing so is w.r.t. decidability and complexity of reasoning, focussing on conjunctive query answering in particular.

## 2 Background and Literature Review

In the following subsections, we give a few basic preliminaries describing Datalog\textsuperscript{\pm}, as well as an overview over the known results in the area.

### 2.1 Background

In this section the basic notions of conjunctive query evaluation under tuple generating dependencies (TGDs) are recalled, including a review of the chase procedure, an important algorithmic tool in the evaluation of queries under TGDs. Furthermore we briefly introduce the concept of stable models in the logic programming perspective. We assume that the reader is familiar with first-order logic as well as basic complexity theory. Good introductions to the former can be found in e.g. \cite{Barwise 1977} and \cite{Andrews 2002}, for the latter we recommend \cite{Papadimitriou 1994}.

#### 2.1.1 Conjunctive Queries and the Relational Model

In order to define the semantics of conjunctive queries, we first need to introduce the relational data model. In the relational data model, the structure or schema \( \mathcal{S} \) of a database and its contents or instance \( D \) are distinct objects.

A schema \( \mathcal{S} \) consists of a finite number of relation symbols (also called predicates) \( r_i \), that is, \( \mathcal{S} = \{ r_1, \ldots, r_n \} \).

Such a relation symbol \( r_i \in \mathcal{S} \) (for any \( i \)) consists of a finite number of attributes, such that each attribute has a domain of possible values. We consider here only the case that all predicates have a common domain \( \Gamma \cup \Gamma_N \), where \( \Gamma \) is a set of constants and \( \Gamma_N \) is a set of labelled nulls (i.e.,
distinct null values, each with a unique name, comparable to skolem constants). The number of
attributes of a relation symbol is called the arity, denoted \( \text{arity}(r_i) \).

A relation \( R_i \) for predicate \( r_i \) is a set of tuples and each tuple is a mapping of each attribute in
\( r_i \) to \( \Gamma \cup \Gamma_N \). Such a tuple of \( R_i \) is denoted by \( r_i(x_1, \ldots, x_k) \) (also referred to as an atom), where
\( k = \text{arity}(i) \).

An instance \( I \) for a schema \( \mathcal{S} \) consists of relations \( R_i \) for each \( r_i \in \mathcal{S} \), that is, \( D = \{ R_1, \ldots, R_n \} \). An
instance in which no null values from \( \Gamma_N \) appear is referred to as a database, usually denoted \( D \). Note that, when viewed as a first-order theory, we may simply interpret an instance as a
conjunction of atoms.

A conjunctive query \( q \) over a database schema \( \mathcal{S} \) is an assertion of the form
\[
q(\vec{X}) \leftarrow \exists \vec{Y} \varphi(\vec{X}, \vec{Y})
\]
where \( \vec{X} \) and \( \vec{Y} \) are vectors of (first-order logic) variables, \( q(\vec{X}) \) is called the head, \( \text{dimen}\text{sion}(\vec{X}) \) is called the arity of \( q \) and \( \varphi(\vec{X}, \vec{Y}) \) is called the body, where \( \varphi(\vec{X}, \vec{Y}) \) is a first-order formula consisting of a conjunction of atoms of the form \( r_i(t_1, \ldots, t_k) \) and equalities of the form \( t_1 = t_2 \), where \( r_i \) is a predicate of \( \mathcal{S} \) with arity \( k \) and each \( t_i \) is either a constant from \( \Gamma \) or a (first-order logic) variable. If the arity is 0 then \( q \) is called a boolean conjunctive query.

With every database \( D = \{ R_1, \ldots, R_n \} \) over a schema \( \mathcal{S} \), we can now associate a finite first-order structure \( M_D = (U, R_1, \ldots, R_n) \) with universe \( U = \Gamma \). The evaluation of a conjunctive query \( q \) then comes down to checking satisfiability in first-order logic as follows: \( q \) has an answer over \( D \), denoted \( D \models q \), if and only if the set \( \{ \{a_1, \ldots, a_k\} | M_D \models q(a_1, \ldots, a_k) \} \) is non-empty, with \( a_i \in \Gamma \). This set is also called the set of answers to \( q \) over \( D \), where \( k \) is the arity of \( q \).

2.1.2 Dependencies

For reasoning tasks over databases, the need arises to express how new (intensional) knowledge can be derived from the data that is stored in the database (called the extensional data). An
established way to do this is to introduce a set \( \Sigma \) of rules that describe the relation between intensional and extensional data. In this case for a database \( D \), the logical theory \( D \cup \Sigma \), i.e., the
conjunction of the facts in the database with all the rules in \( \Sigma \), is taken as a basis for conjunctive
query evaluation.

Rules in \( \Sigma \) over a schema \( \mathcal{S} \) are of either one of the following two forms:
\[
\forall \vec{X}(\varphi(\vec{X}) \rightarrow \exists \vec{Y} \psi(\vec{X}, \vec{Y})) \quad (1)
\]
\[
\forall \vec{X}(\varphi(\vec{X}) \rightarrow X_i = X_j) \quad (2)
\]
where rules of the form of (1) are referred to as tuple generating dependencies (TGDs) and of
(2) as equality generating dependencies (EGDs), with \( \varphi \) and \( \psi \) being conjunctions of predicates
from \( \mathcal{S} \) (also called atoms) and \( X_i \) and \( X_j \) are the \( i \)-th and \( j \)-th position in vector \( \vec{X} \). \( \varphi \) is also
referred to as the body of the dependency and \( \psi \) or \( X_i = X_j \) as the head. TGDs where \( \psi = \bot \) are
called negative constraints. For brevity, we will omit the universal quantifiers in front of TGDs
and EGDs, and replace conjunctions in the body by commas.

Given an instance \( I \), it is said to be satisfying a dependency \( \sigma \in \Sigma \), that is, \( I \models \sigma \), if the first-order
sentence formed by a conjunction of the facts in \( I \) and \( \sigma \) is satisfiable. By extension, \( I \)
satisfies \( \Sigma \) (i.e., \( I \models \Sigma \)) iff it satisfies every \( \sigma \in \Sigma \).

The models of a database \( D \) over a schema \( \mathcal{S} \) with respect to \( \Sigma \), denoted \( \text{Mod}(D, \Sigma) \), are all
instances \( M \) that satisfy \( D \cup \Sigma \) (i.e., \( I \supseteq D \) and \( I \models \Sigma \)). When answering conjunctive queries we
use the certain answer semantics, i.e., we consider the query to be true only if it is true under every model. The set of answers for a conjunctive query \( q \), denoted \( \text{ans}(q, D, \Sigma) \), thus equals the set

\[
\{ \langle a_1, \ldots, a_k \rangle \mid \forall M \in \text{Mod}(D, \Sigma) : M \models q(a_1, \ldots, a_k) \}
\]

For complexity analysis we focus on the decision version of this problem. This is the central problem when analyzing the complexity of databases, tuple and equality generating dependencies and therefore Datalog\(^+\) complexity issues. Below it is formulated for boolean conjunctive queries, which we will focus on in this work:

**BCQ-ANSWERING**

**Instance:** \( \langle q, D, \Sigma \rangle \): \( q \) a boolean conjunctive query, \( D \) a database and \( \Sigma \) a set of dependencies

**Question:** \( D \cup \Sigma \models q \)?

Usually when dealing with query evaluation over databases the **data complexity** and the **combined complexity** are of interest. In this paper we follow the approach of (Vardi 1982) where for the former everything except the database \( D \) is considered fixed, i.e., the only input is the database. For the latter, the database \( D, \Sigma \) and the query itself form the input.

Unfortunately, in general it holds that BCQ-ANSWERING is undecidable under unrestricted sets of TGDs, as has been shown in (Beeri and Vardi 1981). In (Cali et al. 2013; Baget et al. 2009; Baget et al. 2011) it has further been shown that even singleton sets of TGDs cause query answering to become undecidable.

These results clearly show that restrictions must be placed on the structure of \( \Sigma \) to ensure decidability. This is a non-trivial problem, as simple restrictions, like limiting the number of TGDs, are not enough.

### 2.1.3 The Chase

One of the fundamental tools to algorithmically check implication of dependencies is the *chase procedure*, introduced in (Maier et al. 1979), which was later adapted for checking query containment in (Johnson and Klug 1984), in the setting of databases with tuple and equality generating dependencies, or, more specifically, in the setting of databases with inclusion and functional dependencies. The chase algorithm tries to extend a given database instance in such a way that every TGD and EGD becomes satisfied. This is done by exhaustively (i.e., until a fix-point is reached) applying the *chase step*:

**Definition 2.1**

Let \( D \) be a database and \( \Sigma \) be a set of dependencies. A **chase step** is defined as follows:

**TGDs.** Let \( \Sigma \) contain a TGD \( \varphi(\overline{X}) \rightarrow \exists \overline{Y} \psi(\overline{X}, \overline{Y}) \), such that

- \( D \models \varphi(\overline{a}) \) for some assignment \( \overline{a} \) to \( \overline{X} \), and
- \( D \not\models \exists \overline{Y} \psi(\overline{a}, \overline{y}) \).

Then extend \( D \) with facts \( \psi(\overline{a}, \overline{y}) \), where the elements of \( \overline{y} \) are fresh labelled nulls (i.e., values from \( \Gamma_N \) that have not been in use in \( D \) up to that point).

**EGDs.** Let \( \Sigma \) contain an EGD \( \varphi(\overline{X}) \rightarrow X_i = X_j \), such that
\[ D \models \varphi(\vec{a}) \] for some assignment \( \vec{a} \) to \( \vec{X} \), and
\[ a_i \neq a_j \]

If \( a_i \) is a labelled null, then replace every occurrence of \( a_i \) with \( a_j \) or vice-versa if \( a_j \) is a labelled null. If \( a_i \) and \( a_j \) are distinct constants, end the chase with failure.

Definition 2.2
The chase expansion of a database instance \( D \) with respect to a set of dependencies \( \Sigma \) is a sequence \( D_0, D_1, \ldots, D_m \), such that \( D_0 = D \) and for \( i \geq 0 \), \( D_{i+1} \) is obtained from \( D_i \) by applying a chase step. After exhaustively applying such chase steps, we obtain \( D_m \), also denoted \( \text{chase}(D, \Sigma) \).

The chase can have three different outcomes: Failure, non-terminating success or terminating success. In case of success the resulting instance \( D_m \) satisfies all dependencies in \( \Sigma \). Note that if the chase does not terminate, \( m = \infty \) and the size of \( D_m \) is infinite.

We assume that the chase is fair, i.e., we exclude the possibility of a degenerated chase expansion by assuming that the chase expansion is constructed level by level, and after each application of a TGD, all applicable EGDs are applied. This ensures that every TGD that can be applied, is applied, and therefore we exclude the case that only a single infinite path in the chase expansion is ever expanded when in case the chase is infinite.

Query Answering and the Chase
In case the chase succeeds, it computes a universal solution for \( \langle D, \Sigma \rangle \). Every model \( M \in \text{Mod}(D, \Sigma) \) can then be obtained by appropriate instantiation of labelled nulls in \( \text{chase}(D, \Sigma) \) (i.e., for every model \( M \), there exists a homomorphism mapping the universal solution to \( M \); cf. \cite{Deutsch et al. 2008}). Using this property, the chase expansion of a database \( D \), with respect to a set of dependencies \( \Sigma \), can be used for answering conjunctive queries, as the following theorem shows:

Theorem 2.3 \cite{Deutsch et al. 2008}
Given a boolean conjunctive query \( q \) over a schema \( \mathcal{S} \), a database \( D \) of \( \mathcal{S} \) and a set of dependencies \( \Sigma \) over \( \mathcal{S} \), then in cases where the chase does not fail, it holds that \( D \cup \Sigma \models q \) if and only if \( \text{chase}(D, \Sigma) \models q \).

In case the chase fails, query answering is trivial: As there is no model, every boolean conjunctive query clearly is entailed by \( D \cup \Sigma \) (cf. the definition of certain answers in section 2.1.2).

2.2 Literature Review
In this section we discuss the different kinds of restrictions known to ensure decidability of query answering under sets of TGDs. The decidable classes of TGDs discussed below are defined by syntactic properties that either apply to single TGDs (local syntactic conditions) or to the set of all TGDs (global syntactic conditions). These properties can be checked in finite time using appropriate algorithms. Each subsection deals with a known syntactic condition that ensures decidability of query answering.

Inclusion Dependencies
Inclusion dependencies (IDs), one of the simplest forms of dependencies, allow one to express that certain values occurring in a specific position in one relation, must also occur at (or be included in) a specific position in another relation. This allows for TGDs
that consist of one body and head atom only, and no variable may occur twice in the head or the body. The following is an example of an inclusion dependency, expressing that every student is a person:

\[ \text{student}(X,Y) \rightarrow \exists Z \text{person}(X,Z) \]

The query answering problem was shown to be decidable, and in fact in AC\(_0\) (resp. PSPACE) in the data (resp. combined) complexity.

**Linear Tuple Generating Dependencies** This class is similar to IDs in that it allows for TGDs with only a single body atom, but generalizes them, because it allows repetition of variables in the body or head (e.g., the TGD \(r(X,Y,X) \rightarrow s(X,Y)\) is a linear TGD but not an ID).

Sets of linear TGDs enjoy the so-called bounded derivation-depth property (BDDP), which roughly implies that only a finite initial part of the chase is required for query answering, thus ensuring decidability. As with inclusion dependencies, first-order rewritability (i.e., rewriting \(q\) and \(\Sigma\) into a first-order query \(q_\Sigma\), such that \(D \cup \Sigma \models q_\Sigma\) iff \(D \models q\)) is thus possible (cf. (Cali et al. 2009; Cali et al. 2010)). Therefore we get decidability, and query answering is in AC\(_0\) in the data complexity. Regarding combined complexity, results from inclusion dependencies carry over to linear TGDs, resulting in the PSPACE-completeness for query answering in the general case and NP-completeness in case of a fixed set of TGDs.

**Guarded Tuple Generating Dependencies** In (Cali et al. 2013), linear TGDs are extended to so-called guarded TGDs, that have a body atom that contains all variables occurring in the body, i.e., all universally quantified variables. This atom is called the guard. If there are multiple such atoms, the leftmost is taken as the guard. An example of a guarded TGD that says that if students are in their first semester, they have a tutor, is as follows. Note that it is not linear as it has multiple atoms in the body.

\[ \text{student}(X,Y), \text{firstsemester}(X) \rightarrow \exists Z \text{tutor}(X,Z) \]

Linear TGDs and inclusion dependencies are trivially guarded, as they only have exactly one body atom. However, guarded TGDs are not first-order rewritable. This is shown by creating a database, query and a set of guarded TGDs in such a way that answering the query requires the computation of the transitive closure over a relation in the database. It is well known that this property cannot be expressed in a finite first-order query, and we cannot obtain decidability thusly. However, it can be shown that the universal model constructed by the chase, albeit possibly infinite, is of finite treewidth (i.e., it is tree-like and cannot be arbitrarily cyclic). From Courcelle’s famous Theorem (cf. (Courcelle 1990)), which states that evaluating first-order sentences over structures of finite treewidth is decidable, we derive decidability for query answering under sets of guarded TGDs.

The complexity of query answering under guarded TGDs was investigated in (Cali et al. 2009), where it was established that, whenever a query is actually entailed by a database and a set of guarded TGDs, then all atoms needed to answer the query are derived in a finite, initial portion of the chase when restricted only to guards and atoms derived from them, whereby the size of this portion depends only on the query and the set of TGDs. Therefore, constructing this part of the chase and evaluating a boolean conjunctive query over it is enough to compute the answer. It is shown that this can be done in polynomial time in the data complexity, whereby P-membership follows. Hardness for P was shown in (Dantsin et al. 2001) by reduction to the implication problem for propositional logic programs.
The combined complexity is investigated in (Calì et al. 2013), proving the 2Exp-completeness for the general case and Exp-completeness in case of fixed arity. Also membership in NP was shown in case where the set of TGDs is fixed. NP-hardness follows from results in (Chandra and Merlin 1977), which show that NP-hardness holds even for the empty set of TGDs.

**Weakly-Guarded Sets of TGDs** In (Calì et al. 2013), guarded TGDs were extended to weakly-guarded sets of TGDs. Every TGD in such sets must have an atom in its body that contains all the variables where a null value may appear during the chase. The leftmost such atom is called the weak-guard. This class is the first class discussed here that is based on a global property. It is easy to see that, as guarded TGDs contain a body atom with all universally quantified variables, they are trivially weakly-guarded, as the guard is also a weak-guard.

It is implicit in (Calì et al. 2013) that it can be verified in polynomial time whether a set of TGDs is weakly-guarded or not: For a schema \( S \) we first need to compute all the positions for each predicate where a null value can occur during the chase with respect to a set of TGDs \( \Sigma \). These positions are called affected and computing them has been shown to be possible in polynomial time. Then we have to check for each TGD in \( \Sigma \) whether it contains a weak-guard, which, knowing the affected positions, is also possible in polynomial time.

It is then shown that weakly-guarded sets of TGDs enjoy the same favorable property as guarded TGDs, namely, the chase has finite treewidth. Given this fact, decidability of query answering is established as before. Regarding the complexity, in general the problem is 2Exp-complete, Exp-complete if the arity is fixed or the set of TGDs fixed, and it remains Exp-complete even if only the database is considered as input (data complexity).

**Weakly-Acyclic Sets of Tuple Generating Dependencies** The notion of Weak Acyclicity was established in the landmark paper (Fagin et al. 2005) as a syntactic condition to guarantee termination of the chase procedure. For this we first need to define the notion of a dependency graph.

A dependency graph \( G = (V,E) \) is constructed as follows: \( V \) is the set of attributes of all the relations occurring in \( \Sigma \). We will denote the \( i \)th attribute of some relation \( r \) by \( r[i] \). For each TGD \( \sigma = \phi(X) \rightarrow \exists Y \psi(X,Y) \) and each variable \( X \in X \) shared between the relation attributes \( r[i] \) in \( \phi \) and \( s[j] \) in \( \psi \), we add an edge \((r[i],s[j])\) to \( E \). We add a special edge \((r[i],p[k])\) to \( E \) for each attribute \( p[k] \) in \( \psi \) occupied by a variable \( Y \in Y \), and each attribute \( r[i] \) occurring in the body of \( \sigma \).

A set \( \Sigma \) of TGDs is called weakly-acyclic if its dependency graph contains no cycles through special edges. The definition of weak acyclicity is a global property and can be decided in \( P \), as the construction of the dependency graph and the cycle-check through a special edge are both feasible in \( P \).

In (Fagin et al. 2005) it was shown that for weakly-acyclic sets of TGDs the chase always terminates. This is ensured by the fact that when cycles through special edges in the dependency graph are forbidden, no new null values can be added in a later chase step because of a null value added in an earlier chase step. Therefore we trivially get decidability: Simply compute the (finite) chase, and then answer the query on the obtained finite model.

Regarding complexity (cf. (Calì et al. 2013; Kolaitis et al. 2006)), in general the problem of BCQ-ANSWERING is 2Exp-complete for weakly-acyclic sets of TGDs. When the set of TGDs is fixed, the BCQ-ANSWERING problem is known to be NP-complete. P-completeness holds for the data complexity, following from the complexity of the fact inference problem for fixed Datalog programs (see (Dantsin et al. 2001)).
Sticky Sets of Tuple Generating Dependencies
A recent addition to the set of syntactic conditions that ensure decidability and favourable complexity of conjunctive query evaluation is the paradigm of stickiness, introduced in (Cali et al. 2012). A survey of sticky classes can be found in (Cali et al. 2010). The class of sticky sets of TGDs is defined as follows: In a first step, a variable marking of all TGDs in a set Σ is computed by a procedure called SMarking. This is a two-step procedure:

1. Initial marking: For each σ ∈ Σ, if there exists a variable V in the body of σ and an atom without this variable exists in the head of σ, mark each occurrence of V in the body.

2. Propagation step: Until a fixpoint is reached, consider any pair (σ1, σ2) ∈ Σ × Σ. If a universally quantified variable V occurs in head(σ1) at positions π1, . . . , πm for m ≥ 1 and an atom in body(σ2) exists where at each of these same positions a marked variable occurs, then mark each occurrence of V in body(σ1).

Definition 2.4 (Cali et al. 2012)
A set Σ of TGDs is called sticky if and only if there is no TGD in SMarking(Σ) such that a marked variable occurs in its body more than once.

The property of stickiness is incomparable to guardedness and weak acyclicity but strictly generalizes inclusion dependencies. In comparison to other discussed syntactic classes of TGDs, sticky sets of TGDs allow for a mildly restricted way to express joins. The following is an example of a sticky (singleton) set of TGDs, expressing the join between two tables, department and employee, to get a combined table of departments and their heads:

\[ \text{department}(X, Y), \text{employee}(Y, Z) \rightarrow \text{headofdept}(X, Y, Z) \]

Note that the above TGD is not weakly-guarded.

3 Goal and Current Status of the Research
The goal, as already discussed in the introduction, is to introduce disjunction into Datalog\(^±\) and investigate the impact of doing so on the decidability and complexity of query answering. We thus extend the definition of a TGD to allow for disjunction as follows:

A disjunctive tuple-generating dependency (DTGD) σ is a first-order formula \( \forall X \varphi(X) \rightarrow \bigvee_{i=1}^{n} \exists Y \psi_i(X, Y) \), where \( n \geq 1 \), \( X \cup Y' \subseteq \Gamma_\varphi \), and \( \varphi, \psi_1, \ldots, \psi_n \) are conjunctions of atoms; \( \varphi \) is the body of \( \sigma \), denoted body(σ), while \( \bigvee_{i=1}^{n} \psi_i \) is the head of \( \sigma \), denoted head(σ). If \( n = 1 \), then \( \sigma \) is a tuple-generating dependency (TGD). Given a set \( \Sigma \) of DTGDs, schema(\( \Sigma \)) is the set of predicates occurring in \( \Sigma \).

We employ the disjunctive chase introduced in (Deutsch and Tannen 2003) in order to answer queries. It is an extension of the chase procedure described in Section 2.1. Consider an instance I, and a DTGD \( \sigma = \varphi(X) \rightarrow \bigvee_{i=1}^{n} \exists Y \psi_i(X, Y) \). We say that \( \sigma \) is applicable to I if there exists a homomorphism \( h \) (i.e., a substitution of labelled nulls to either constants or other labelled nulls) such that \( h(\varphi(X)) \subseteq I \), but there is no \( i \in \{1, \ldots, n\} \) and a homomorphism \( h' \supseteq h \) such that \( h'(\psi_i(X, Y)) \subseteq I \). The result of applying \( \sigma \) to I with \( h \) is the set \( \{I_1, \ldots, I_n\} \), where \( I_i = I \cup h'(\psi_i(X, Y)) \), for each \( i \in \{1, \ldots, n\} \), and \( h' \supseteq h \) is such that \( h'(Y) \) is a “fresh” labelled null not occurring in I, for each \( Y \in Y' \). For such an application of a DTGD, which defines a single DTGD chase step, we write \( I(\sigma, h) \{I_1, \ldots, I_n\} \).

A disjunctive chase tree of a database D and a set \( \Sigma \) of DTGDs is a (possibly infinite) tree such
that the root is $D$, and for every node $I$, assuming that $\{I_1, \ldots, I_n\}$ are the children of $I$, there exists $\sigma \in \Sigma$ and a homomorphism $h$ such that $I(\sigma, h)\{I_1, \ldots, I_n\}$. The disjunctive chase algorithm for $D$ and $\Sigma$ consists of an exhaustive application of DTGD chase steps in a fair fashion, which leads to a disjunctive chase tree $T$ of $D$ and $\Sigma$; we denote by $\text{dchase}(D, \Sigma)$ the set $\{I \mid I \text{ is a leaf of } T\}$. Note that each leaf of $T$ is well-defined as the least fixpoint of a monotonic operator. By construction, each instance of $\text{dchase}(D, \Sigma)$ is a model of $D$ and $\Sigma$. Interestingly, $\text{dchase}(D, \Sigma)$ is a universal set model of $D$ and $\Sigma$, i.e., for each $M \in \text{Mod}(D, \Sigma)$, there exists $I \in \text{dchase}(D, \Sigma)$ and a homomorphism $h_I$ such that $h_I(I) \subseteq M$ [Deutsch et al. 2008]. This implies that w.r.t. certain answers, given a query $q$, $D \cup \Sigma \models q$ iff $I \models q$, for each $I \in \text{dchase}(D, \Sigma)$.

**Current Status.** Currently we have investigated and obtained results for all the decidable classes of TGDs. For the guarded-based classes, adding disjunction does not make the problem of query answering undecidable. However it does in certain cases increase the complexity of the problem by a significant amount. For the guarded-based classes of TGDs (i.e., IDs, linear, guarded and weakly-guarded), we have established all relevant complexity results when extending them to DTGDs.

In case of sticky TGDs, when adding disjunction the problem of query answering becomes undecidable. This was a very surprising result, given the fact that the complexity of query answering under sticky sets of TGDs is lower than under guarded TGDs.

In case of weakly-acyclic TGDs, data complexity results have been obtained, as well as certain lower bounds in the combined complexity, however a matching upper bound is still missing here. Decidability is assured in any case, because the disjunctive chase terminates, which follows from the definition of weak acyclicity.

**4 Preliminary Results**

One classical work on disjunction in ontologies is [Calvanese et al. 2006], which immediately gives us coNP-hardness for conjunctive query answering over disjunctive ontologies, even if the query is fixed, and the ontology consists of a fixed, single rule of the form $a(X) \rightarrow b(X) \lor c(X)$. Without restricting the query language, there is thus no hope to get tractability results. However, for atomic queries, where the query consists only of a single atom, there are tractable data complexity cases to be found.

**Arbitrary queries.** In [Bourhis et al. 2013], we have investigated the complexity picture for answering arbitrary queries. The main results are as follows:

- $2\text{EXP}$-completeness whenever the query is non fixed. This is shown by simulating a Büchi tree automaton, and it even holds for fixed sets of Disjunctive Inclusion Dependencies (DIDs) of arity at most three, or of non-fixed sets of the same with arity at most two.
- coNP-completeness in the data complexity for query answering under DIDs up to guarded DTGDs.
- EXP-completeness in the data complexity for query answering under weakly-guarded sets of DTGDs.

In case of (non-disjunctive, classical) TGDs, complexity results coincide in the data complexity, but vary from coNP-completeness to $2\text{EXP}$-completeness for fixed sets of IDs to weakly-guarded sets of TGDs. It is thus interesting to note that adding disjunction to expressive languages
doesn’t change the complexity in this case, but there is a high cost to add it to less expressive languages.

Atomic queries. In [Gottlob et al. 2012], we have investigated the complexity of answering single-atom queries. Here the complexity results vary considerably:

- 2Exp-completeness in the combined complexity for guarded DTGDs.
- EXP-completeness in the combined complexity for linear DTGDs.
- coNP-completeness in the data complexity for guarded DTGDs.
- Membership in \( \mathbf{AC_0} \) in the data complexity for linear DTGDs.

In the case of atomic queries we do have a number of tractability results to offer, especially the highly parallelizable data complexity of \( \mathbf{AC_0} \) in case of atomic query answering over sets of linear DTGDs (which captures the class of DIDs). For guarded DTGDs, most of the results follow directly from expressive fragments of first-order logic (the Guarded Fragment [Bárány et al. 2010; Grädel 1999], and Guarded-Negation First-Order Logic [Bárány et al. 2011; Bárány et al. 2012]). For linear, we develop novel machinery to obtain our respective bounds.

5 Open Issues and Expected Achievements

In addition to the published results, we would like to find answers to the following questions:

What is the complexity of query answering under sets of

1. guarded-based DTGDs in case where the query is acyclic or of bounded (hyper)treewidth?
2. weakly-acyclic sets of DTGDs?
3. sticky sets of DTGDs?

Regarding the first item, we have already managed to obtain all the relevant results. In fact, for bounded (hyper)treewidth, the complexity table coincides with that of arbitrary queries. For acyclic queries, there are drops in complexity corresponding to the expressivity of the language considered. Papers containing these results have been submitted to this year’s MFCS conference and DL workshop. It is our plan to subsequently publish these results, in addition to some extended work on arbitrary and atomic queries in a comprehensive journal paper, treating all the guarded-based classes of DTGDs, in the course of 2014.

Regarding weakly-acyclic, we already have answers to the complexity questions for data complexity and the cases of fixed sets and fixed arities. However, we are still missing the combined complexity results. Before submission of my thesis, we plan to close these open complexity questions as well.

Lastly, for sticky DTGDs, we have an undecidability proof, which is somewhat surprising as query answering under sticky TGDs is easier in terms of complexity than it is for guarded TGDs, yet the addition of disjunction doesn’t cause a complexity increase in the latter. We have therefore focused on extending guarded DTGDs with cross-products (a form of join allowed in sticky TGDs). This again yields undecidability, however it becomes decidable if restricted to arity at most two, where binary predicates can never participate in a disjunction. For this case we are working on obtaining the relevant complexity results.
Reasoning under Disjunctive Existential Rules

References


MICROSOFT CORP. 2011. The microsoft connected services framework.


CDF-Intervals: A Reliable Framework to Reason about Data with Uncertainty

AYA SAAD
Universität Ulm, Germany
(e-mail: ayas@aucegypt.edu)

submitted 7 March 2014; revised 7 April 2014; accepted 15 May 2014

Abstract

This research introduces a new constraint domain for reasoning about data with uncertainty. It extends convex modeling with the notion of p-box to gain additional quantifiable information on the data whereabouts. Unlike existing approaches, the p-box envelops an unknown probability instead of approximating its representation. The p-box bounds are uniform cumulative distribution functions (cdf) in order to employ linear computations in the probabilistic domain. The reasoning by means of p-box cdf-intervals is an interval computation which is exerted on the real domain then it is projected onto the cdf domain. This operation conveys additional knowledge represented by the obtained probabilistic bounds. Empirical evaluation shows that, with minimal overhead, the output solution set realizes a full enclosure of the data along with tighter bounds on its probabilistic distributions.

KEYWORDS: convex structures, reliable constraint reasoning, probability box, cdf interval, constraint satisfaction problem, constraint programming, constraint reasoning, uncertainty

1 Introduction

This research proposes a novel constraint domain for reasoning about data with uncertainty. The work was driven by the practical usage of reliable approaches in Constraint Programming (CP). These approaches tackle large scale constraint optimization (LSCO) problems associated with data uncertainty in an intuitive and tractable manner. Yet they have a lack of knowledge when the data whereabouts are to be considered. These whereabouts often indicate the data likelihood or chance of occurrence, which in turn, can be ill-defined or have a fluctuating nature. It is important to know the source and type of the data whereabouts in order to reason about them. The purpose of this novel framework is to intuitively describe data coupled with uncertainty or following unknown distributions without losing any knowledge given in the problem definition. The p-box cdf-intervals extend the cdf-intervals approach, (Saad et al. 2010), with a p-box structure to obtain a safe enclosure. This enclosure envelops the data along with its whereabouts with two distinct quantile values, each is issuing a cdf-uniform distribution, (Saad et al. 2012). This research is concerned with the following contributions: (1) a new uncertain data representation specified by p-box cdf-intervals, (2) a constraint reasoning framework that is used to prune variable domains in a p-box cdf-interval constraint relation to ensure their local consistency, (3) an experimental evaluation, using the inventory management problem, to compare the novel framework with existing approaches in terms of expressiveness and tractability. The expressiveness, in this comparison, measures the ability to model the uncertainty provided in the original problem, and the
impact of this representation on the solution set realized. On the other hand, the tractability measures the system time performance and scalability. The experimental work shows how this novel domain representation yields more informed results, while remaining computationally effective and competitive with previous work.

2 Preliminaries

Models tackling uncertainty are classified under the set of plausibility measures (Halpern 2003). They are categorized as: possibilistic and probabilistic. Convex models, found in the world of fuzzy and interval/robust programming, are favored when ignorance takes place. They are adopted in the CP paradigm in fuzzy Constraint Satisfaction Problems (CSPs) (Dubois et al. 1996) and numerical CSPs (Benhamou and de Vinci 1994). Probabilistic models are best adopted when the data has a fluctuating nature. They are the heart of the probabilistic CP modeling found in valued CSP (Schiex et al. 1995), semirings (Bistarelli et al. 1999), stochastic CSPs (Walsh 2002), scenario-based CSPs (Tarim et al. 2006) and mixed CSPs (Fargier et al. 1996). Techniques adopting convex modeling are characterized to be more conservative. They can often consider many unnecessary outcomes along with important ones. This conservative property supplements convex modeling with a high tractible and scalable behavior since operations, in these models, are exerted on the convex bounds only. On the other hand, probabilistic approaches add a quantitative information that expresses the likelihood, yet these approaches impose assumptions on the distribution shape in order to conceptually deal with it in a mathematical manner. Moreover, probabilistic mathematical computations are very expensive because they often depend on the non-linear probability shape. The research objective is to introduce a novel framework: the p-box cdf-intervals. It is based on a probability box (p-box) structure (Ferson et al. 2003) that envelops a set of cumulative distribution functions (cdf). The p-box concept is adopted in the literature, specifically when the environment is uncertain, to represent an unknown distribution with a safe enclosure rather than depending on statistical approximation. A cdf is a monotone (non-decreasing) function that indicates for a given uncertain value the probability that the actual data lies before. It defines the aggregated probability of a value to occur. The p-box bounding cdf distributions in the proposed framework are uniform, each is represented by a line equation in order to maintain an inexpensive computational complexity. The key idea behind the construction of the p-box cdf-intervals is to combine techniques from the convex models, to take advantage of their tractability, with approaches revealing quantifiable information from the probabilistic and stochastic world, to take advantage of their expressiveness.

The framework is based on CP concepts because they proved to have a considerable flexibility in formulating real-world combinatorial problems. The CP paradigm aims at building descriptive algebraic structures which are easily embedded into declarative programming languages. These structures are heavily used in problem solving environment by specifying conditions that need to be satisfied and allow the solver to search for feasible solutions. The following section demonstrates how to intuitively represent the uncertainty, already given in the problem definition, in order to reason about it by means of the p-box cdf-intervals. A comparison between the novel representation of the data uncertainty with existing possibilistic and probabilistic approaches is also taking place in order to demonstrate the model expressiveness. This representation is input to the solver with a new domain specification. Consequently the reasoning about this new specification is defined. It proves how the reasoning by means of p-box cdf-intervals is tractable.
Accordingly, combining reasoning techniques from convex models with quantifiable information from probabilistic models yields a novel model that is together tractable and expressive.

3 Input Data Representation

Quantifiable information is often available during the data collection process, but lost during the reasoning because it is not accounted for in the representation of the uncertain data. This information however is crucial to the reasoning process, and the lack of its interpretation yields erroneous reasoning because of its absence in the produced solution set. It is always necessary to quantify uncertainty that is naturally given in the problem definition in order to obtain robust and reliable solutions.

Example 3.1. Consider, as a running example, the varying cost observations of a steel stud manufacturing item. Fig. 1(a) illustrates the cost variations along with their corresponding frequencies of occurrence. For instance, the point $(5.17, 4)$ is the amount of the cost/item, equal to 5.17, and observed 4 times during the past 2 years (corresponding to a population $m = 40$). 9 is the number of distinct measured quantiles. The minimum and the maximum observed values, in this example, are 5.17 and 6.36 respectively.

To compute the probabilistic/possibilistic representations, the average and standard deviation of the observed population are derived. In this example, they are equal to 5.6 and 0.28 respectively. The nearest Normal probability distribution and the fuzzy membership function are illustrated in Fig. 1 (b) and (c). To compare the data representations adopted in various approaches, the observed data is projected onto the cdf-domain. By definition, the cdf is a monotonic distribution that keeps the probabilistic information in an aggregated manner. Information obtained from the measurement process is often discrete and incomplete, hence, its cdf-domain projection forms a staircase shape (Smith and La Poutre 1992). The cdf distribution of the genuine observed data whereabouts is depicted in the running example by the dotted staircase shape in Figure 2. Normal and fuzzy cdf distributions are shown by the continuous red curves in Fig. 2 (b) and (c). Each is based on an approximation that lacks precise point fitting of the original data whereabouts. Similarly, the cdf-interval, in Fig. 2 (d), approximates the data whereabouts by means of a line connecting the two bounding data values. The convex model representation however shapes a rectangle, illustrated in Fig. 2 (e). This rectangle includes all values in the cdf range $[0, 1]$. The convex representation treats data values lying within the interval bounds equally, i.e. it lacks the probabilistic information. The p-box cdf-interval, depicted in Fig. 2 (f) enforces tighter bounds on the probabilities when compared to convex models illustrated in Fig. 2 (e). This envelopment guarantees a safe enclosure on the unknown distribution while preserving tractability due to the fact that its bounds are represented each by a line equation.
Interpretation of the p-box cdf confidence interval $I$. For a given interval of points specified by $I = [p_a, p_b]$, $p_a$ and $p_b$ are the extreme points which bound the p-box cdf-interval. One can see that this interval approach does not aim at approximating the curve but rather enclosing it in a reliable manner. The complete envelopment is exerted by means of the uniform $cdf$-bounds, which are depicted by the red curves in Fig. 2 (f). It is impossible to find a point that exists outside the formed interval bounds. The $cdf$ bounds are chosen to have a uniform distribution. Each is represented by a line with a slope issued from one of the extreme quantiles. Storing the full information of each bound is sufficient to restore the designated interval assignment. Bounds are denoted by triplet points, in the 2D space, to guarantee the full information on: the extreme quantile value observed; the $cdf$-line issued from this observed value; and the degree of steepness formed by this line. The slope of the uniform $cdf$-distribution indicates how the probabilistic values accumulate for successive quantiles on the line. Accordingly, the p-box $cdf$-interval point representation: $p_a = (a, F_{p_a}, S_{p_a})$ and $q_b = (b, F_{q_b}, S_{q_b})$.

Definition 3.1. $S_{p_a}$ is the slope of a given $cdf$-distribution; it signifies the average step probabilistic value. For a given uniform $cdf$-distribution

\[ S_{p_a} = \frac{F_{p_b} - F_{p_a}}{b - a}, \forall a \leq x \leq b \tag{1} \]

Plotting a point $p_x$ within the p-box $cdf$-interval deduces bounds on its possible chances of occurrence.

Definition 3.2. $F^I_x$ is the interval of values obtained when $p_x$ is projected onto the p-box $cdf$ bounds. For a point $p_x \in I$ denoted as $p_x = (x, F_{p_x}, S_{p_x})$

\[
    a < x < b, \text{ and } F^I_a \leq F^I_x \leq F^I_b \text{ and } S^I_a \leq S^I_x \leq S^I_b \tag{2}
\]

$F^I_a$ and $F^I_b$ are the possible maximum and minimum $cdf$ values $p_x$ can take; both are computed by projecting the point $p_x$ onto the $cdf$ distributions passing through real points $a$ and $b$ respectively. They are derived using the following linear projections, computed in $O(1)$ complexity:

\[
    F^I_a = \min(S^I_a(x - a) + F_{p_a}, 1) \quad \text{and} \quad F^I_b = \max(F_{p_b} - S^I_b(b - x), 0) \tag{3}
\]

The equation above guarantees the probabilistic feature of the $cdf$-function by restricting its aggregated value from exceeding the value 1 and having negative values below 0.
Example 3.2. \( I = [(5.17, 0.1, 1.2), (6.36, 0.7, 0.57)] \) is the p-box \( \text{cdf} \)-interval of the cost/item in Example 3.1. Suppose that \( x_i = 5.5 \), its \( \text{cdf} \)-bound values \( F^I_i = [0.2, 0.5] \). This means that the possible chance of the value to be at most 5.5 is between 20% and 50%, with an average step probabilistic value between 0.57 and 1.2. Note that this interval is opposed to only one approximated value \( F_i = 0.37 \) in the \( \text{cdf} \)-intervals representation proposed in (Saad et al. 2010), the \textit{fuzzy cdf} value \( F_i = 0.31 \) and its Normal \( \text{cdf} \) value is \( F_i = 0.42 \). Note that convex models do not enforce any probabilistic bounds, accordingly, \( x_i = 5.5 \) has a \( \text{cdf} \) \( F^I_i \in [0, 1] \).

4 Constraint reasoning

In the CP paradigm, relations between variables are specified as constraints. A set of rules and algebraic semantics, defined over the list of constraints, formalize the reasoning about the problem. As a fundamental language component in the Constraint Logic Programming (CLP), these set of rules, with a syntax of definite clauses, form the language scheme (Jaffar and Lassez 1987). The constraint solving scheme is intuitively and efficiently utilized in the reasoning over the computation domain. The scheme formally attempts at assigning to variables a suitable do-

\begin{itemize}
  \item \text{Equality constraints:} \( \{ =, \neq \} \), and that of the ternary arithmetic constraints \( \{ +, -, \times, \div, \wedge, \vee \} \). Operations, in the solver, are exerted first as real interval computations, and then they are projected onto the \( \text{cdf} \) domain using a linear computation, as shown in Definition 3.2. This section demonstrates how the ordering and the ternary addition constraints infer the local consistency over the variable domains of \( X, Y \), and \( Z \) assuming that their initial bindings are \( I = [p_a, p_b] \), \( J = [q_c, q_d] \) and \( K = [r_e, r_f] \) respectively. The ternary multiplication, subtraction and division constraints are implemented in the same way.

\begin{itemize}
  \item \text{Ordering constraint} \( X \leq Y \). To infer the local consistency of the binary ordering constraint, the lower \( \text{cdf} \)-bound of \( X \) is extended and the upper \( \text{cdf} \)-bound of \( Y \) is contracted.

Example 4.1. Let \( I \) and \( J \) be two p-box \( \text{cdf} \)-interval domains. \( I = [(10, 0.14, 0.016), (80, 0.49, 0.06)] \) and \( J = [(20, 0.06, 0.025), (90, 0.9, 0.014)] \). The effect of applying the set of constraints \( X \geq I \) and \( X \leq J \), prunes the domain of \( X \). As a result, the variable \( X \) is bounded by the lower bound of \( I \) and by the upper bound of \( J \). \( X \in [(10, 0.14, 0.016), (90, 0.9, 0.014)] \) as shown in Fig. 3 (a). Clearly the obtained domain of \( X \), in this example, preserves the convex property of the p-box.
Let $Y$ be subject to the domain pruning using the set of constraints: $Y \preceq I$ and $Y \succeq J$. As a result, $Y$ should be bounded by the lower bound of $J$ and the upper bound of $I$. However, in this case, at lower quantiles $\leq 23$, the upper bound distribution of $I$ precedes the lower bound of $J$. The fact that conflicts the stochastic dominance property of a p-box $cdf$-interval domain. In order to resolve this conflict, the real bounds of $Y$ are further pruned to the point of the probability intersection $= 23$.

**Fig. 3: Ordering constraint execution**

**Ternary addition constraints** $X + \gamma Y = Z$. The addition operation is implemented by summing up pair of points, defined in the 2D space and located within the p-box $cdf$-interval bounds which enclose the domain ranges of $X$ and $Y$. This addition operation is linear. It is convex and can be computed from the end points of the domains involved in the addition. The p-box $cdf$-domain of $Z$ is updated to envelop all points defined in that range.

**Example 4.2.** Fig. 4 depicts the execution steps of the p-box $cdf$ ternary addition inference rule, exerted on the variable domains involved in the relation $Z = X + \gamma Y$. Observe that domain pruning is performed in a 2 dimensional manner: quantile and $cdf$. The addition of the two variables $X$ and $Y$ is performed on the bounds of their predefined domains then it is projected onto the initial bindings. The first row in Fig. 4 shows output domains from the addition $I + J$, $K − J$ and $K − I$. Domain operations are exerted on the extreme points. The second row illustrates the intersection of the output domains with the initial bindings, assigned to $Z$, $X$ and $Y$. Obtained domains from the ternary addition operation are $K'$, $I'$ and $J'$. Clearly, in this example, pruning real quantile bounds is identical to that of real domains and since output domains preserve the stochastic dominance property no further pruning takes place.

The ternary addition constraint exerted on p-box $cdf$-interval domains is a simple addition computation since it adopts the real-interval arithmetics which are then projected, linearly, onto the $cdf$ domain. This operation is opposed to the fuzzy extended addition operation adopted in the constraint reasoning utilized in the possibilistic domain (Dutta et al. 2005; Petrović et al. 1996),
Fig. 4: Ternary addition inference rule execution: initial bindings are $X \in I$, $Y \in J$, and $Z \in K$; final bindings are $X \in I'$, $Y \in J'$, and $Z \in K'$.

and to the Normal probabilistic addition which has a high computation complexity that is due to the Normal distribution shape (Glen et al. 2004).

5 Empirical evaluation

The inventory management problem model proposed by (Tarim and Kingsman 2004) is employed, as a case study, to evaluate the proposed framework. The key idea is to schedule ahead replenishment periods and find the optimal order sizes which achieve a minimum total manufacturing cost. A reorder point with order size $X$ should meet customer demands up to the next point of replenishment.

Definition 5.1. An inventory management model defined over a time horizon of $N$ cycles is

$$
\text{minimize} \quad TC = \sum_{t=1}^{N} (a\delta_t + hI_t + vX_t) \\
\text{subject to} \quad \delta_t = \begin{cases} 
1 & \text{if } X_t > 0 \\
0 & \text{otherwise}
\end{cases} \\
I_t = I_0 + \sum_{t=1}^{N} (X_t - d_t) \\
X_t, I_t \geq 0, \ t = 0, 1, \ldots, N
$$

The constituents of the total cost in the model are: the setup cost, holding cost and purchase cost. The setup cost is defined by the ordering cost multiplied by the number of times a replenishment takes place. The holding cost depends on the depreciation cost and the level of the inventory observed in a given cycle. The purchase cost is the reorder quantity multiplied by the varying cost/item. From this model, one can observe that all cost components are typically fluctuating and unpredictable especially in the real-life version of the problem. This is due to
the unpredictability of customer demands and the variability of the cost/item. Accordingly, this model perfectly fits the purpose of the evaluation: comparing the behavior of the models when the environment is uncertain.

**Information realized in the solution set.** The model is tested for a randomly distributed monthly demands over a time horizon \( N = 10 \) cycles. The p-box \( \text{cdf} \)-interval representation is constructed for each demand observation per cycle and for each observed varying cost component (ordering cost \( a \), holding cost per item \( h \) and varying cost per item \( v \)) to guarantee a safe enclosure on the data whereabouts. This is opposed to the fuzzy and probabilistic modeling which is based on the average demand values given in the set \( d_i \in \{26, 36, 23, 28, 32, 30, 37, 25, 34\} \). The two later models set assumptions on the shape of the probability distribution adopted, as pointed out in Section 3. The solver executes the set of addition and equality constraints in the p-box \( \text{cdf} \)-interval domain. Constraints are triggered until stabilized and consistency is reached by means of the inference rules defined in Section 4. The solver suggests 2 to 5 replenishment periods, with a total holding cost \([8.5, 0.83, 4.4E-04], (137.98, 0.039, 7.5e-05)\] and a total manufacturing cost \([2739.6, 0.8, 3.3E-04], (6483.2, 0.03, 6.2e-05)\]. This output is opposed to 6 replenishment periods realized by the fuzzy and the probabilistic models with a total holding cost $53.5 and $52.05 and a total manufacturing cost $3868.5 and $3828.93 respectively. Fig. 5 illustrates a comparison between the output holding cost obtained from the models under consideration. The p-box \( \text{cdf} \)-interval graphical representation of the cost is depicted by the shaded region and their bounds in the convex models are illustrated by the dotted rectangles. Clearly, the solution set obtained from the p-box \( \text{cdf} \)-intervals model, when compared with the outcome of the convex model, realized an additional knowledge (i.e. tighter bounds in the \( \text{cdf} \) domain). This solution set is opposed to a one value proposed as $53.5 by the fuzzy and as $52.05 by the probabilistic models. Output solution point suggested by the latter models can, sometime, mislead or deviate the decision making. This is because their distributions are built, from the beginning, on approximating the actual observed distribution.

**Model tractability.** We generate random distributions for monthly demands scaling up the problem time horizon for \( \{7, 10, 24\} \) cycles. The first three rows in Table 1 show the real time taken by each model in seconds to generate the output solution of the total cost. Two other measurements, the shared heap used and the control stack used, are taken into consideration in order to study the memory consumption of each model. The shared heap used is the memory allocated to store compiled Prolog code and its related variables and necessary buffers. The control stack used is utilized to hold backtracking information. Table 1 demonstrates that stochastic model memory consumption grows exponentially when scaling-up the problem, it reaches 100% of the memory usage for a time horizon \( t = 24 \). The p-box \( \text{cdf} \)-intervals behavior is similar to convex models.
Probabilistic and fuzzy models have the best shared heap utilization. Clearly the percentage of the control stack employed in the stochastic model is the highest. This is due to the behavior of the stochastic techniques which exhaustively build the solution scenarios in order to reach a solution. It is worth noting that convex models and p-box cdf-intervals do not need to build this tree since output solution set is provided within an interval range that is encapsulating all possible output scenarios. Evidently, convex models outperform the rest of the models in terms of speed; p-box cdf-intervals have a closer speed, followed by the fuzzy models, then by the probabilistic models. In summary, the p-box cdf-intervals performance is closer to that of the convex models. This means that, the new framework, with minimal overhead, adds up a quantifiable information by imposing tighter bounds on the probability distribution, in a safe and in a tractable manner. Applied computations are tractable because they are exerted on the interval bounds, using interval computations, then results are further projected, linearly, onto the cdf domain. Empirical evaluations proved that p-box cdf-intervals have a scalability measure that is close to that of convex models.

### 6 Conclusion and future research direction

This research proposes a novel constraint domain to reason about data with uncertainty. The key idea is to extend convex models with the notion of p-boxes in order to realize additional quantifiable information on the data whereabouts. P-Boxes have never been implemented in the CP paradigm, yet they are very good candidates to deal with and reason about uncertainty in the probabilistic paradigm, especially when data is shaping an unknown distribution. The case study of the inventory management problem demonstrates that p-box cdf-intervals can be practically adopted to intuitively envelop the uncertain data found in different modeling aspects with minimum overhead. Evaluation results show that stochastic CPs and probabilistic models have the slowest performance. Fuzzy models proved to have a better performance and their output solutions are characterized to be reliable, i.e. they seek the satisfaction of all possible realizations. Convex models and the p-box cdf-intervals encapsulate all possible distributions of the solution set in a convex representation. The p-box cdf-intervals framework provides a range of quantiles along with bounds on their data whereabouts.

The introduction of a novel framework to reason about data coupled with uncertainty due to ignorance or based on variability, paves the way to many fruitful research directions. We can list many in: studying models having variables following dependent probability distributions, exploring different search techniques, revisiting the framework within a dynamically changing

<table>
<thead>
<tr>
<th>time horizon $t$</th>
<th>stochastic</th>
<th>probabilistic</th>
<th>fuzzy</th>
<th>cdf</th>
<th>p-box</th>
<th>convex</th>
</tr>
</thead>
<tbody>
<tr>
<td>real time (sec)</td>
<td>7</td>
<td>0.43</td>
<td>3.71</td>
<td>3.06</td>
<td>0.81</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>70.14</td>
<td>6.08</td>
<td>5.77</td>
<td>3.28</td>
<td>3.28</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>1683.36</td>
<td>175.22</td>
<td>159.05</td>
<td>55.41</td>
<td>32.5</td>
</tr>
<tr>
<td>shared heap used</td>
<td>7</td>
<td>0.21%</td>
<td>0.4%</td>
<td>0.29%</td>
<td>6.87%</td>
<td>6.86%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.21%</td>
<td>0.5%</td>
<td>0.29%</td>
<td>9.68%</td>
<td>9.67%</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>100%</td>
<td>0.9%</td>
<td>0.7%</td>
<td>22.93%</td>
<td>23.04%</td>
</tr>
<tr>
<td>control stack used</td>
<td>7</td>
<td>62.87%</td>
<td>46.71%</td>
<td>23.35%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>89.82%</td>
<td>46.71%</td>
<td>23.35%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>100%</td>
<td>46.71%</td>
<td>23.35%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 1: Real-time taken and measurement of memory consumption
A. SAAD

environment, generalizing the framework to deal with all types of uncertainty by considering together vagueness and dynamicity, and last but not least applying the model to a variety of large scale optimization problems which target real-life engineering and management applications.

References


Visualization of Constraint Handling Rules

Nada Sharaf¹, Slim Abdennadher¹ and Thom Frühwirth²
¹ The German University in Cairo, Egypt; ² Ulm University, Germany
(e-mail: {nada.hamed, slim.abdennadher}@guc.edu.eg, thom.fruehwirth@uni-ulm.de)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Constraint Handling Rules (CHR) has matured into a general purpose language over the past two decades. Any general purpose language requires its own development tools. Visualization tools, in particular, facilitate many tasks for programmers as well as beginners to the language. The article presents on-going work towards the visualization of CHR programs. The process is done through source-to-source transformation. It aims towards reaching a generic transformer to visualize different algorithms implemented in CHR.

Note: An extended abstract / full version of a paper accepted to be presented at the Doctoral Consortium of the 30th International Conference on Logic Programming (ICLP 2014), July 19-22, Vienna, Austria.

KEYWORDS: Constraint Handling Rules, Algorithm Visualization, Source-to-Source Transformation

1 Introduction

Although Constraint Handling Rules (CHR) [Frühwirth 1998] was introduced as a language for writing constraint solvers, it has developed into a general purpose language over the years. CHR is a committed choice language. A CHR program consists of multi-headed guarded rules. In CHR, predicates are transformed into simpler ones until they are solved. CHR has a number of implementations. However, the most prominent ones are embedded in Prolog.

Since CHR has developed into a general purpose language, CHR programmers can now write programs to implement general algorithms such as sorting algorithms, graph algorithms, etc. With such algorithms, programs could get very long and in some cases complicated. Thus development tools and especially tracing and visualization tools are very useful and sometimes even necessary. As discussed in [Hundhausen et al. 2002], algorithm visualization technologies are useful in many cases such as in practical laboratories, for in-class discussions, or in assignments where students could for example produce their own visualizations. It could help instructors find bugs quickly. Moreover, such visualizations could be useful for debugging and tracing the implementations of different algorithms.

There are different methods for embedding visualization features into a CHR program. One of them is to alter the compiler or the CHR runtime system. This solution is however not recommended since performing such changes is not an easy task for any programmer.
especially a beginner. Thus the adopted approach is to use source-to-source transformation to eliminate the need of doing any changes to the running system.

Program transformation or source-to-source transformation, allows developers to add or change the behavior of programs without manually modifying the initial code. In addition, according to (Loveman 1977), source-to-source transformation could be useful for improving the performance of different programs.

In (Abdennadher and Sharaf 2012), a first attempt towards visualizing CHR programs was presented. The presented tool was able to visualize the execution of CHR programs was realized through source-to-source transformation. In addition, it was able to visualize CHR constraints as objects. The system, however, lacked generality and required ad-hoc hard-wired inputs. In addition visualizing algorithms implemented through the different CHR programs was not possible. Some systems (Schulte 1997; Simonis and Aggoun 2000) provided visualization options for constraint programs. However, the focus was on the search space and trees rather than the executed algorithms. This paper thus presents a new approach that aims at visualizing CHR solvers in a generic way overcoming the problems of the old tool. Through the new system, various algorithms implemented through CHR could be visualized.

The final goal of the project is to have a general source-to-source transformation tool that is able to automatically add different extensions to CHR solvers.

The paper is organized as follows: Section 2 introduces CHR through an example. Section 3 discusses in more details the suggested architecture of the system. An example of the output of the system is shown in Section 4. Finally, some conclusions and directions for future work are shown in Section 5.

### 2 Constraint Handling Rules

This section presents an example of a CHR program to introduce the syntax and semantics of CHR (Frühwirth 2009). In CHR, two types of constraints are available. The first type is the built-in constraints that are provided through the host language. The second type of constraints is the CHR or user-defined constraints that are defined through the rules of the program. A CHR program consists of simpagation rules with the form:

\[
\text{name} @ H_k \setminus H_r \iff G \mid B.
\]

The name of the rule precedes the @ sign and is optional. The head of the CHR rule, comes before the (\(\iff\)). It should only contain a conjunction of CHR constraints. As seen from the previous rule, there are two parts in the head namely \(H_k\) and \(H_r\). \(H_k\) contains the constraints that are kept after the rule is executed. However, the constraints in \(H_r\) are removed after executing the rule. The guard \((G)\) is optional and should only contain built-in constraints. Finally, the body \((B)\) could contain both CHR and built-in constraints. The constraints in the body are added to the constraint store on executing the rule. A rule is executed only if the head constraints match some of the constraints in the constraint store (Frühwirth 2008). In addition, the guard has to be satisfied for the rule to be executed. At the beginning of the execution, the constraint store is empty. It is initialized by the constraints in the query. There are special cases of simpagation rules which are simplification and propagation rules. Whenever \(H_k\) is empty, the resulting rule is a “simplification” rule. The head of a
simplification rule contains CHR constraints that are removed once the rule is executed. Thus through simplification rules, CHR constraints are replaced by simpler ones. Thus the format of a simplification rule is:

\[ H_r \leftrightarrow G \mid B. \]

In propagation rules, \( H_r \) is empty. Consequently, the rule does not remove any constraints from the store. It only adds the constraints in the body to the store. Propagation rules have the following format:

\[ H_k \Rightarrow G \mid B. \]

The following example is for a CHR program that sorts numbers. The numbers are fed into the solver using the constraint \texttt{list}. For example, the constraint \texttt{list(I,V)} means that the cell at index \( I \) of the list has the value \( V \). The solver contains the simplification rule \texttt{sortlist}.

\texttt{sortlist @ list(Index1,V1), list(Index2,V2) \iff Index1<Index2 , V1>V2 \mid list(Index2,V1), list(Index1,V2).}

The rule \texttt{sortlist} makes sure that a number precedes another one in the list if and only if it is indeed smaller than it. If this is not the case, the two numbers are swapped. Consequently, applying such a rule results in a sorted sequence of numbers. For example if the input to the solver is \texttt{list(0,7), list(1,6), list(2,4)}, execution proceeds as follows:

\[
\begin{align*}
\text{list}(0,7), \text{list}(1,6), \text{list}(2,4) \\
\quad \downarrow \\
\text{list}(1,7), \text{list}(0,6), \text{list}(2,4) \\
\quad \downarrow \\
\text{list}(1,7), \text{list}(2,6), \text{list}(0,4) \\
\quad \downarrow \\
\text{list}(2,7), \text{list}(1,6), \text{list}(0,4)
\end{align*}
\]

As seen from the previous execution sample, every time a new number is added to the sequence or the store, the program makes sure it is placed in the correct position with respect to the already existing elements. Accordingly, after all numbers are added, the resulting sequence is a sorted one. The semantics implemented in SWI Prolog is the refined operational semantics [Duck et al. 2004]. It makes sure that constraints are processed from the left to the right and that rules are executed in a top-bottom approach as demonstrated through the previous example. For example after \texttt{list(2,4)} is added to the constraint store, the rule \texttt{sortlist} is executed since 6 and 4 are not sorted in the sequence. Thus, \texttt{list(0,6)} and \texttt{list(2,4)} are removed from the constraint store. They should be replaced by \texttt{list(2,6)} and \texttt{list(0,4)}. However, these two constraints are added to the store one by one. Once \texttt{list(2,6)} is added to the store and even before adding \texttt{list(0,4)}, the rule \texttt{sortlist} is executed using the two constraints \texttt{list(2,6)}, \texttt{list(1,7)}. A more detailed view of the execution steps is:
Nada Sharaf, Slim Abdennadher and Thom Frühwirth

An Empty Constraint Store

\[ \text{list}(0,7) \]

Adding \text{list}(0,7) to the store

\[ \text{list}(1,6), \text{list}(0,7) \]

Adding \text{list}(1,6) to the store

\[ \text{sortlist} \text{ removing list}(0,7), \text{list}(1,6) \text{ and adding list}(1,7) \text{ as a first step} \]

\[ \text{list}(1,7) \]

Adding \text{list}(1,7) to the store

\[ \text{list}(0,6), \text{list}(1,7) \]

Adding \text{list}(0,6) to the store

\[ \text{sortlist} \text{ removing list}(0,7), \text{list}(1,6) \text{ and adding list}(1,7) \text{ as a first step} \]

\[ \text{list}(2,4), \text{list}(0,6), \text{list}(1,7) \]

Adding \text{list}(2,4) to the store

\[ \text{sortlist} \text{ removing list}(2,4), \text{list}(0,6) \text{ and adding list}(2,6) \text{ as a first step} \]

\[ \text{list}(2,6), \text{list}(1,7) \]

Adding \text{list}(2,6) to the store

\[ \text{sortlist} \text{ removing list}(2,4), \text{list}(0,6) \text{ and adding list}(2,6) \text{ as a first step} \]

\[ \text{list}(2,7) \]

Adding \text{list}(2,7) to the store

\[ \text{sortlist} \text{ removing list}(2,6), \text{list}(1,7) \text{ and adding list}(2,7) \text{ as a first step} \]

\[ \text{list}(1,6), \text{list}(2,7) \]

Adding \text{list}(1,6) to the store

\[ \text{list}(0,4), \text{list}(1,6), \text{list}(2,7) \]

Adding \text{list}(0,4) to the store

3 System Architecture

This section introduces the adopted architecture and transformation approach. The tool presented by (Abdennadher and Sharaf 2012) was able to add visualization features to CHR programs. The tool however lacked the possibility of visualizing different algorithms implemented through CHR. In order to extend the tool, the user was always required to enter specific hard-wired inputs. The focus is now for a new and a more general approach. As shown in Figure 1 the general architecture of the workbench consists of several modules. At the beginning, the CHR program is fed into the parser. In addition to parsing the input file, the parser also extracts the needed information and represents it in the required format for the transformer.

The transformer uses the “relational normal form” presented in (Frühwirth and Holzbaur 2003). This form uses some special CHR constraints to encode the different constituents of a CHR rule. Such constraints include head, body and guard. The parser thus represents the information about the different rules using the specified form. For example, head(sortlist,’list(Index1,V1)’,remove) represents the fact that the rule sortlist has the constraint list(Index1,V1) in its head. In addition, this constraint is removed on executing the rule since it is a simplification rule. Finally, the new solver is generated by the transformer. Unlike the form presented in (Frühwirth and Holzbaur 2003), the transformer neglects the identifier of the head constraint since it is not needed.
In addition, the system makes use of a new module called the “Annotation Module” explained in more detail in Section 3.1.

The output programs are normal CHR programs that could run with SWI-Prolog. Generally, when the output solver is running, log files are generated as shown in Figure 1. These files should contain information regarding the executed rules that could then be used by an optional external module.

The external module is utilized by the visualization extension. This module reads the log files generated by the new program. In addition it makes use of the output of the Annotation Module in order to produce the visualization file needed by the visual tracer. This visual tracer could be any visualization program. For proof of concept we used Jawaa (Pierson and Rodger 1998).

### 3.1 Annotation Module

This module was proposed as a step towards eliminating the need of entering algorithm-specific information. The new approach this paper introduces also aimed at a more general visual tracer that does not need to be changed every time the algorithm differs. The idea is to have a more basic visualization tool and a more intelligent transformation process. Thus, we decided to out-source the actual visualization process to existing tools such as Jawaa, OpenSCAD\(^1\) etc. Such systems provide different sets of visualization objects and possibly actions as well. Nevertheless, the need of connecting the transformed CHR programs to these systems in a generic way to be able to visualize any algorithm remained. As introduced in (Kerren and Stasko 2002), algorithm animation or software visualization produces abstractions for the data and the operations of an algorithm. The different states of the algorithm are represented as images that are animated according to the different interactions between such states.

\(^1\) [http://www.openscad.org/](http://www.openscad.org/)
The adapted idea in the new system is to visualize the CHR constraints themselves as objects. After transforming the initial program, the rules of the new program modify the constraints and thus the objects. Consequently, the execution of each rule adds one step to the visual tracer. Viewing the sequence of objects thus produces an animation showing how the rules acted on the constraints (or objects) and thus visualizes the algorithm. Accordingly, the “Annotation Module” was suggested. The module extends the system with the annotation functionality which allows it to deal with different visualization tools while keeping a general scheme. When deploying the system, users enter the annotations or the mappings between the different constraints and the objects provided through the visualization system. Figure 2 shows how the user mapped the constraint *list/2* to the visual object “Node” that Jawaa offers. In addition to stating the name of the object, the user also states how the arguments of the constraints affect the parameters needed for the object. The parameters for the *Node* object include the name, the x-coordinate, the y-coordinate, the width, the height in addition to the text. The values the user enters for these parameters are also associated with the annotations. The values could use some of the arguments of the constraints. As shown in Figure 2, the constraint is mapped into a *Node* object which means that everytime a *list/2* constraint is added to the store a new *Node* is visualized. The name of the *Node* is chosen to be *nodeValueOf( Value)* which means that the each element in the list will have a node with a name corresponding to its value. For example, the Node corresponding to the number 7 is a *node7*. The x-coordinate is calculated with the following formula *valueOf(Index) * 12 + 2. Through this formula the Node of the element at index 0 is placed at the x-coordinate 2. The Node of the second element (with index 1) is placed at the x-coordinate 14. The third element

![Figure 2: Annotating CHR constraints.](image-url)
is placed at 26. Given that the width is determined to be 10, this means that there is a 2 pixels gap between any two consecutive nodes. The y-coordinate is determined to be 50 for all of the Nodes. The height, on the other hand, is five times the value of the element represented through $\text{valueOf}(\text{Value}) \times 5$. The node contains only one piece of information which is the value of the element. The outline color of the node is black and the background color is green.

Whenever the user saves the annotation, the following XML file is produced keeping track of all of the associations to the constraint.

```xml
<association>
  <constraint name="list (Index, Value)">
    <add name="node" parameters="name=nodevalueOf(arg1)\#x=valueOf(arg0) +12\#y=50\#
      width=10\#height=valueOf(arg1)\times5\#n=1\#data=valuef(Value)\#color=black#
      bkgrd=green\#textcolor=black\#type=RECT" type="arg1"/>
  </constraint>
</association>
```

### 3.2 Transformation Module

This section shows how the rules in the original CHR program are transformed to be able to interact with the visualization system. Source-to-source transformation was used in order to avoid any manual changes. The basic scheme is similar to the one shown in [Abdennadher and Sharaf 2012](#). In more details, the new solver interacts with an external module. This module was implemented in Java. The external module uses the information propagated through the solver in addition to the saved annotations to produce a file that could be animated through the visualization system such as Jawaa or OpenSCAD.

To have a step-by-step animation, the execution of each rule communicates the needed information.

In general any CHR rule of the form:

\[
\text{rule } @ \ H_k \ \ H_r \ \langle = \rangle \ G \ \mid \ \text{Body.}
\]

is transformed to a rule of the form:

\[
\text{rule } @ \ H_k \ \ H_r \ \langle = \rangle \ G \ \mid \ \text{communicate} \_\hk(H_k), \ \text{communicate} \_\hr(H_r), \ \text{Body.}
\]

As seen from the scheme, the functionality of the rule is kept intact. However, the auxiliary predicate `communicate/1` is used within the new body of the rule. The aim of this predicate is to send the head of the rule to the external module. This module can then start to act upon the information to add the next visualization step.

Actually, only the heads that are removed from the constraint store can affect the visualization. In such a case, the visualized objects for these constraints should be removed from the visualization window.

In some cases, the constraints of the head will not affect the visualization. The transformer can then be instructed to produce a new solver that does not communicate to the external module the head constraints. In such output solvers the old rules are kept intact:

\[
\text{rule } @ \ H_k \ \ H_r \ \langle = \rangle \ G \ \mid \ \text{Body.}
\]
However, such transformation is not sufficient since the constraints in the body of the rule were not communicated. The body-constraints are responsible for adding new constraints/objects to the trace. Since with the proposed system each constraint maps to an object, a generic way of solving this problem is to add for each constraint \( \text{cons}/n \) a rule with the form:

\[
\text{cons}(\text{arg}_1, \text{arg}_2, \text{arg}_3, \ldots, \text{arg}_n) \Rightarrow \text{communicate}(\text{cons}(\text{arg}_1, \text{arg}_2, \text{arg}_3, \ldots, \text{arg}_n)).
\]

The previous rule is a propagation rule which does not affect the constraint store. It is however triggered once a constraint of the form \( \text{cons}(\text{arg}_1, \text{arg}_2, \text{arg}_3, \ldots, \text{arg}_n) \) is added to the store communicating to the external module the new constraint in the store. If this constraint has a corresponding annotation, the module adds the needed visualization step. In addition, if a rule adds multiple constraints that have corresponding annotations, they will be animated one by one. Thus the visualization step is done once the constraint is added to the store. Such propagation rules are added at the beginning of the program to make sure they are executed once a constraint is added and before executing any other applicable rule.

4 Jawaa Example

This section shows how the sorting algorithm shown in Section 2 is visualized using Jawaa. The output solver has the following extra rule at its beginning:

\[
\text{list}(\text{V0}, \text{V1}) \implies \text{communicate}(\text{list}(\text{V0}, \text{V1})).
\]

In addition the rule \text{sortlist} is modified such that it communicates to the external module its head constraints since their corresponding objects need to be removed. Thus the new rule has the following format:

\[
\text{sortlist} @ \text{list}(\text{Index1}, \text{V1}), \text{list}(\text{Index2}, \text{V2}) \iff \text{Index1}<\text{Index2}, \text{V1} > \text{V2} \mid \\
\text{communicate}_\text{hr}(\text{list}(\text{Index1}, \text{V1})), \text{communicate}_\text{hr}(\text{list}(\text{Index2}, \text{V2})), \text{list}(\text{Index2}, \text{V1}), \text{list}(\text{Index1}, \text{V2}).
\]

To have an animation through Jawaa, a “anim” file that contains all of the animation details needs to be used. The external module makes use of the information communicated from the new solver in addition to the saved annotations in order to generate the corresponding animation file. The external module is able to dynamically build up the animation file step by step.

For the query \text{list}(7,0), \text{list}(6,1),\text{list}(4,2), the generated animation file after processing the query is shown in Appendix A. As seen from the file, each time a new \text{list} constraint was generated, the corresponding node was added. The first generated constraint \text{list}(0,7) adds to the file:

\[
\text{node node7 2 50 10 35 1 7 black green black RECT}
\]

which adds a node with x-coordinate: 2, y-coordinate: 50, width: 10, height 35. The text in the node is 7. The execution of the solver keeps on adding \text{list} constraints and thus Jawaa nodes are added. In addition, everytime two elements are swapped, their corresponding nodes are removed since the rule \text{sortlist} is a simplification rule. The resulting animation is shown in Figure 3.
Visualization of Constraint Handling Rules

Fig. 3: Sorting the sequence 7, 6, 4 using the sorting solver introduced in Section 2.

<association>
<constraint name="list(Index, Value)"
<add name="text" parameters="name=nodevalueOf(arg1)#x=valueOf(arg0)*12+2#y=50#text=valueOf(arg1)#color=black#size=30" type="Object"/>
</constraint>
</association>

Figure 4 shows the resulting animation using the query list(0,7), list(1,6), list(0,4).

5 Conclusion

The paper introduced a transformation approach that is able to add visualization features to CHR solvers. The new system overcomes the drawbacks of the old approach. As seen, the need for explicit inputs is removed. The system was able to map constraints into generic existing objects. There was also no need to change the compiler or the runtime system. We are currently in the process of producing a portable version of the application. It will provide users with running examples in addition to patterns for the annotation module. Throughout the paper, Jawaa was used for proof of concept. However, for the future, the system will also be tested with different tools such as OpenSCAD. This approach can be used with an existing visualization system since the intelligence is moved to the transformer and not the tracer. The possibility of using the system to visualize
Fig. 4: Sorting the sequence 7, 6, 4 using the sorting solver introduced in Section 2.

different CHR semantics should be examined. The final goal is to have a generic transformation workbench for CHR. The possibility of having data about the annotations should be also studied. In such a case, the combination of constraints could be used to fire an event.

References


Appendix A  Jawaa Animation File

delay 2500
begin
node node7 2 50 10 35 1 7 black green black RECT
end
delay 2500
begin
node node6 14 50 10 30 1 6 black green black RECT
end
delay 2500
begin
remove node7
remove node6
end
delay 2500
begin
node node7 14 50 10 35 1 7 black green black RECT
end
delay 2500
begin
node node6 26 50 10 30 1 6 black green black RECT
end
delay 2500
begin
node node4 26 50 10 20 1 4 black green black RECT
end
delay 2500
begin
remove node6
remove node4
end
delay 2500
begin
node node6 26 50 10 30 1 6 black green black RECT
end
delay 2500
begin
remove node7
remove node6
end
delay 2500
begin
node node7 26 50 10 35 1 7 black green black RECT
end
delay 2500
Appendix B  Jawaa Animation File using Text Annotation Object

delay 2500
begin
  text node7 2 50 7 black 30
end
delay 2500
begin
  text node6 14 50 6 black 30
end
delay 2500
begin
  remove node7
  remove node6
end
begin
  text node7 14 50 7 black 30
end
delay 2500
begin
  text node6 2 50 6 black 30
end
delay 2500
begin
  text node4 26 50 4 black 30
end
delay 2500
begin
  remove node6
  remove node4
end
begin
  text node6 26 50 6 black 30
end
delay 2500
begin
  remove node7
  remove node6

Application of Methods for Syntax Analysis of Context-Free Languages to Query Evaluation of Logic Programs

Heike Stephan
Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik, 06099 Halle (Saale), Germany
heike.stephan@informatik.uni-halle.de

submitted 15 May 2014; revised 15 May 2014; accepted 15 May 2014

Abstract

My research goal is to employ a parser generation algorithm based on the Earley parsing algorithm to the evaluation and compilation of queries to logic programs, especially to deductive databases. By means of partial deduction, from a query to a logic program a parameterized automaton is to be generated that models the evaluation of this query. This automaton can be compiled to executable code; thus we expect a speedup in runtime of query evaluation.

An extended abstract/full version of a paper accepted to be presented at the Doctoral Consortium of the 30th International Conference on Logic Programming (ICLP 2014), July 19-22, Vienna, Austria

KEYWORDS: Deductive Databases, Earley Deduction, Partial Deduction

1 Introduction and problem description

Deductive databases and related rule-based systems are of increasing interest nowadays (ontologies/semantic web, artificial intelligence, business rules). Especially when large amounts of data have to be processed, an optimal runtime performance is crucial. This can be achieved by compiling the intensional database beforehand, ideally combined with partial evaluation of queries to the deductive database, also known as partial deduction. In this area, lots of work has already been done (see e.g. (Komorowski 1992)). Runtime performance can also be improved by tabling methods, also known as memoing or memoization techniques (e.g. the OLDT method (Tamaki and Sato 1986)), which reuse answers to equivalent subgoals and avoid their recomputation.

One approach that has not yet been taken full advantage of in the past is to make use of the structural similarity of sets of horn clauses, which form classic logic programs, and context-free grammars (see (Ullman 1992) for a good description of this connection). Especially for deductive databases the relationship is quite obvious: Intensional predicates correspond to nonterminal symbols and extensional predicates to terminal symbols of a grammar. A query to the database marks a start point corresponding to the start symbol of a grammar. With this, a query to a logic program can be seen as a call to a context-free
grammar to produce all words of the language. A parser can be easily modified to fulfil
this task instead of consuming words.

The idea is to modify existing, powerful parsing and parser generation algorithms so
that they can process logic programs. The rich knowledge of parser generation is thus of
great value for the task of compiling queries to logic programs: A parser for a deterministic
context-free language can move through a word and decide on its acceptance efficiently
because all grammar derivations that are applicable when a new terminal symbol is
read are compiled into the parser; a similar performance for a compiled query to a logic
program is desirable. The existing query-evaluating or query-compiling methods look
only at single or few similar rules; with applying parser-generation methods, all rules
derivable using a database fact can be processed at once. For this reason, we can expect
to be better than existing query-evaluation/ query-compilation methods if we employ
parsing algorithms for query evaluation.

2 Background and overview of the existing literature

Adapting parsing algorithms to sets of horn clauses has especially been done for def-
inite clause grammars (DCGs)—and so remained in the community of computational
linguistics—but the ideas may principally be transferred to logic programs in general.
One option is to use algorithms related to the LR(k) algorithm developed by Knuth
(Knuth 1965). The LR(k) algorithm is well known and widely used in compiler construc-
tion (for a detailed description, see a textbook on compiler construction as e.g. (Aho
et al. 2007)). An adaptation to DCGs is presented e.g. by Nilsson in (Nilsson 1986).
Here, a logic program is first reduced to its underlying context-free grammar for which
a LR(k) parser can be generated in traditional manner. The predicate arguments are
added during parsing via an argument stack. Algorithms related to the LR(k) algorithm
are attractive because, in contrast to pure top-down algorithms, they can be used on left
recursive grammars. The main drawback is that the LR(k) algorithm heavily relies on
looking ahead one or more characters of the input string in order to guarantee determin-
ism of the generated parser. For the execution of logic programs no input string exists
so that the class of programs for which a deterministic execution model can be created
is quite small. Of course, one can choose to accept nondeterminism and still profit by
being able to cope with left recursion.

Another useful parsing algorithm for evaluating a logic program is Earley Deduction
by Pereira and Warren (Pereira and Warren 1983). This method is inspired by the
Earley Algorithm (Earley 1970), a LR parsing algorithm derived from Knuth’s LR(k)
parser generation algorithm and suitable for all context-free grammars. In contrast to
the LR(k) algorithm, Earley Deduction is completely independent of looking ahead input
characters. This makes Earley Deduction more attractive for the use in query evaluation
than methods based on the LR(k) algorithm.

Porter (Porter III 2009a; Porter III 2009b) tested Earley Deduction in the context
of deductive databases and devised several optimizations for the derivation process. He
introduced the notion of the schema of a rule which makes rules comparable independent
of their actual data values. By this means, rules with the same schema can be collected in
one data structure and indexed for easy access. Additionally, he suggested to precompile
derivation steps to a set of simple instructions which can be applied to sets of rules.
The compilation is done at execution time and depends on the actual data values of the rules with which the comparison or reduction has to be performed. Nevertheless, Porter found that his optimizations significantly increase execution speed compared to naïve Earley Deduction.

3 Goal of the research

The goal of research is to construct an automaton that models the evaluation process for a logic program in a similar way as a parser is generated automatically from a given grammar. The Earley-based methods are less restricted than the LR(k) related algorithms for parser generation and will therefore serve as a base for the method to be developed. The following problems have to be solved on the way to this goal:

1. In contrast to Knuth’s LR(k) algorithm, neither the Earley parsing algorithm nor Earley Deduction are parser generation algorithms. There is no description of algorithms that generate parsers that execute the Earley parsing algorithm or Earley Deduction.

2. In order to have a chance to perform better than existing tabling methods, we have to precompile as much of the query evaluation process as possible. This includes especially those derivations that are possible without retrieving new data from the extensional database. In Knuth’s LR(k) algorithm and in the Earley parsing algorithm, but not in Earley Deduction, the parser uses states that correspond to sets of partially processed grammar productions derivable after reading one terminal from the input string; reading a terminal leads to a state transition. This behaviour is also desirable for Earley Deduction.

3. A parser or parser generator for a DCG—and thus also a query evaluator generator for a logic program—has to cope with the problem of predicate arguments, which are not present in context-free grammars. Not only the status of the evaluation process has to be stored in a state, but also a part of the data from the extensional database. This leads to the unusual concepts of a parameterized state and a parameterized automaton, where the parameters are placeholders for data values retrieved at runtime.

By solving these problems, the state-transition feature of the original Earley Algorithm will be combined with the derivation of rules as in Earley Deduction to construct a generator for an automaton modeling the query evaluation process. So classical concepts of parser generation are employed for obtaining efficient means to compile a logic program into executable code, which implements the generated automaton.

The states of the automaton correspond to sets of rules derivable after reading one fact from the extensional database; input of an extensional database fact leads to a state transition. The idea is to represent such a rule set in a way that maps those parts of the rules that depend only on the program to constant symbols, leaving the data values as variables or parameters that are bound at runtime. The derivation operations are basically the same for different facts of the same extensional relation and can be computed at compile time. Thus several derivation steps can be processed at once and, additionally, reduced to a sequence of variable assignments and to few comparisons.
With this we expect to be able to speed up execution by transferring as many actions as possible to the compilation phase. The execution of a program compiled in this way is expected to be faster than execution with Prolog, the Magic Set method (Bancilhon et al. 1986) or tabling methods, at least in the general case.

4 Preliminary results accomplished

First results have already been published together with my advisor, Prof. S. Brass in (Stephan and Brass 2012) and (Brass and Stephan 2013). The Earley Deduction method is enhanced by a state transitions and by a generator for a parameterized finite automaton. It can be applied to a non-recursive or left-recursive Datalog program. The execution of the generated automaton performs query evaluation on the given extensional database. The states of this automaton can also be viewed as new Datalog predicates and the state transitions as Datalog rules, so we have a rewritten Datalog program. A bottom-up evaluation of this generated program corresponds to the execution of the parameterized automaton.

5 Current status of the research

Currently I am preparing for publication a new version of the method that is able to deal with Datalog programs of the kind of the Same-Generations Program. I also have almost finished thoughts on a version of the method that is applicable to general logic programs, including negation as finite failure/stratified negation.

6 Open issues and expected achievements

There are several ideas on how the versions of the Earley-based compilation methods can be implemented; there will have to be comparisons between those implementations as well as to other established implementations of Datalog and Prolog, especially those using tabling and Magic Sets. The expected improvement of runtime performance must be demonstrated. Furthermore, I am particularly interested in reapplying my compilation method to parser generation and examine how the logic programs that can be processed with the current version are connected to grammars for deterministic context-free languages. If those connections are sufficient, I intend to use the method as parser generator for attributed grammars.

References


Knuth, D. E. 1965. On the translation of languages from left to right. *Information and Control* 8, 6 (Dezember), 607–639.


Reasoning with Probabilistic Logics

RICCARDO ZESE

Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, 44122, Ferrara, Italy
(e-mail: riccardo.zese@unife.it)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

The interest in the combination of probability with logics for modeling the world has rapidly increased in the last few years. One of the most effective approaches is the Distribution Semantics which was adopted by many logic programming languages and in Description Logics. In this paper, we illustrate the work we have done in this research field by presenting a probabilistic semantics for description logics and reasoning and learning algorithms. In particular, we present in detail the system TRILL^P, which computes the probability of queries w.r.t. probabilistic knowledge bases, which has been implemented in Prolog.


1 Introduction

In the last few years, many researchers tried to combine first-order logic and probability for modeling uncertain domains and performing inference and learning. In the field of Probabilistic Logic Programming (PLP for short) many proposals have been presented. An effective and popular approach is the Distribution Semantics (Sato 1995), which underlies many PLP languages such as PRISM (Sato 1995; Sato and Kameya 2001), Independent Choice Logic (Poole 1997), Logic Programs with Annotated Disjunctions (Vennekens et al. 2004) and ProbLog (De Raedt et al. 2007). Along this line, many researchers proposed to combine probability theory with Description Logics (DLs for short) (Lukasiewicz and Straccia 2008; Straccia 2008). DLs are at the basis of the Web Ontology Language (OWL for short), a family of knowledge representation formalisms used for modeling information of the Semantic Web (Hitzler et al. 2009). In (Riguzzi et al. 2012) we presented DISPONTE, a probabilistic semantics for DLs based on the distribution semantics that allows probabilistic assertional and terminological knowledge.

In order to allow inference over the information in the Semantic Web, many efficient DL reasoners, such as Pellet (Sirin et al. 2007), RacerPro (Haarslev et al. 2012) and HermiT (Shearer et al. 2008), have been developed. Despite the availability of many DL reasoners, the number of probabilistic reasoners is quite small. In (Riguzzi et al. 2013a) we presented BUNDLE, a reasoner based on Pellet that extends it by allowing to perform inference on DISPONTE theories. Most of the available DL reasoners, included BUNDLE, exploit procedural languages for implementing their reasoning algorithms. Nonetheless, some of
them use non-deterministic operators for doing inference. We implemented a reasoner, called TRILL (Zese et al. 2013), that exploits Prolog for managing the non-determinism. Then, we developed a new version of TRILL, called TRILL$^P$ and we added in both versions the ability to manage DISPONTE knowledge bases (KBs for short) and computing the probability of a query given a probabilistic KB under the DISPONTE semantics.

Since a problem of probabilistic KBs is that the parameters are difficult to define, in (Riguzzi et al. 2013b) we presented EDGE that learns the parameters of a DISPONTE KB from the information available in the domain. Moreover, we are currently working on the extension of EDGE in order also to learn the structure of the probabilistic KB together with the parameters.

In the field of PLP, we are working at improving existing algorithms. We have considered lifted inference that allows to perform inference in a time that is polynomial in the variables’ domain size. We applied lifted variable elimination, and GC-FOVE (Taghipour et al. 2013) in particular, to PLP and developed the algorithm LP$^2$.

The paper is organised as follows. Section 2 briefly introduces $\mathcal{ALC}$, while Section 3 presents the DISPONTE semantics. Section 4 defines the problem of finding explanations for a probabilistic query w.r.t. a given probabilistic KB. Section 5 presents TRILL and TRILL$^P$ and Section 6 discusses related work. Section 7 shows experiments and section 8 discusses our achievements and future plans. Finally, Section 9 concludes the paper.

2 Description Logics

DLs are knowledge representation formalisms that are at the basis of the Semantic Web (Baader et al. 2003; Baader et al. 2008) and are used for modeling ontologies. They are represented using a syntax based on concepts, basically sets of individuals of the domain, and roles, sets of pairs of individuals of the domain. In this section, we recall the expressive description logic $\mathcal{ALC}$ (Schmidt-Schauß and Smolka 1991). We refer to (Lukasiewicz and Straccia 2008) for a detailed description of $\mathcal{SHOIN(D)}$ DL, that is at the basis of OWL DL.

Let $\mathbf{A}$, $\mathbf{R}$ and $\mathbf{I}$ be sets of atomic concepts, roles and individuals. A role is an atomic role $R \in \mathbf{R}$. Concepts are defined by induction as follows. Each $C \in \mathbf{A}$, $\bot$ and $\top$ are concepts. If $C$, $C_1$ and $C_2$ are concepts and $R \in \mathbf{R}$, then $(C_1 \cap C_2)$, $(C_1 \cup C_2)$, $\neg C$, $\exists R.C$, and $\forall R.C$ are concepts. Let $C$, $D$ be concepts, $R \in \mathbf{R}$ and $a, b \in \mathbf{I}$. An ABox $\mathcal{A}$ is a finite set of concept membership axioms $a : C$ and role membership axioms $(a, b) : R$, while a TBox $\mathcal{T}$ is a finite set of concept inclusion axioms $C \sqsubseteq D$. $C \equiv D$ abbreviates $C \sqsubseteq D$ and $D \sqsubseteq C$.

A knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$. A KB $\mathcal{K}$ is assigned a semantics in terms of set-theoretic interpretations $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ is a non-empty domain and $\cdot^\mathcal{I}$ is the interpretation function that assigns an element in $\Delta^\mathcal{I}$ to each $a \in \mathbf{I}$, a subset of $\Delta^\mathcal{I}$ to each $C \in \mathbf{A}$ and a subset of $\Delta^\mathcal{I} \times \Delta^\mathcal{I}$ to each $R \in \mathbf{R}$.

A query $Q$ over a KB $\mathcal{K}$ is an axiom for which we want to test the entailment from the knowledge base, written $\mathcal{K} \models Q$. The entailment test may be reduced to checking the unsatisfiability of a concept in the knowledge base, i.e., the emptiness of the concept. For example, the entailment of the axiom $C \sqsubseteq D$ may be tested by checking the satisfiability of the concept $C \cap \neg D$. 
3 The DISPONTE Semantics

DISPONTE (Riguzzi et al. 2012) applies the distribution semantics (Sato 1995) of probabilistic logic programming to DLs. A program following this semantics defines a probability distribution over normal logic programs called worlds. Then the distribution is extended to queries and the probability of a query is obtained by marginalizing the joint distribution of the query and the programs.

In DISPONTE, a probabilistic knowledge base $K$ is a set of certain axioms or probabilistic axioms in which each axiom is independent evidence. Certain axioms take the form of regular DL axioms while probabilistic axioms are $p :: E$ where $p$ is a real number in $[0,1]$ and $E$ is a DL axiom.

The idea of DISPONTE is to associate independent Boolean random variables to the probabilistic axioms. To obtain a world, we include every formula obtained from a certain axiom. For each probabilistic axiom, we decide whether to include it or not in $w$. A world therefore is a non probabilistic KB that can be assigned a semantics in the usual way. A query is entailed by a world if it is true in every model of the world.

The probability $p$ can be interpreted as an epistemic probability, i.e., as the degree of our belief in axiom $E$. For example, a probabilistic concept membership axiom $p :: a : C$ means that we have degree of belief $p$ in $C(a)$. A probabilistic concept inclusion axiom of the form $p :: C \sqsubseteq D$ represents our belief in the truth of $C \sqsubseteq D$ with probability $p$.

Formally, an atomic choice is a couple $(E_i,k)$ where $E_i$ is the $i$th probabilistic axiom and $k \in \{0,1\}$. $k$ indicates whether $E_i$ is chosen to be included in a world ($k = 1$) or not ($k = 0$). A composite choice $\kappa$ is a consistent set of atomic choices, i.e., $(E_i,k) \in \kappa$, $(E_i,m) \in \kappa$ implies $k = m$ (only one decision is taken for each formula). The probability of a composite choice $\kappa$ is $P(\kappa) = \prod_{(E_i,1) \in \kappa} p_i \prod_{(E_i,0) \in \kappa}(1 - p_i)$, where $p_i$ is the probability associated with axiom $E_i$. A selection $\sigma$ is a total composite choice, i.e., it contains an atomic choice $(E_i,k)$ for every probabilistic axiom of the probabilistic KB. A selection $\sigma$ identifies a theory $w_{\sigma}$ called a world in this way: $w_{\sigma} = C \cup \{E_i| (E_i,1) \in \sigma\}$ where $C$ is the set of certain axioms. Let us indicate with $S_K$ the set of all selections and with $W_K$ the set of all worlds. The probability of a world $w_{\sigma}$ is $P(w_{\sigma}) = P(\sigma) = \prod_{(E_i,1) \in \sigma} p_i \prod_{(E_i,0) \in \sigma}(1 - p_i)$. $P(w_{\sigma})$ is a probability distribution over worlds, i.e., $\sum_{w \in W_K} P(w) = 1$.

We can now assign probabilities to queries. Given a world $w$, the probability of a query $Q$ is defined as $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. The probability of a query can be defined by marginalizing the joint probability of the query and the worlds, i.e., $P(Q) = \sum_{w \in W_K} P(Q,w) = \sum_{w \in W_K} P(Q|w)p(w) = \sum_{w \in W_K,w|w=Q} P(Q)$.  

**Example 3.1**

Consider the following KB, inspired by the people+pets ontology (Patel-Schneider et al. 2003):

\[
0.5 :: \exists hasAnimal.Pet \sqsubseteq NatureLover \quad 0.6 :: \text{Cat} \sqsubseteq \text{Pet}
\]

\[
(\text{kevin}, \text{tom}) : \text{hasAnimal} \quad (\text{kevin}, \text{fluffy}) : \text{hasAnimal} \quad \text{tom} : \text{Cat} \quad \text{fluffy} : \text{Cat}
\]

The KB indicates that the individuals that own an animal which is a pet are nature lovers with a 50% probability and that kevin has the animals fluffy and tom. Fluffy and tom are cats and cats are pets with probability 60%. We associate a Boolean variable to each axiom as follow $F_1 = \exists hasAnimal.Pet \sqsubseteq NatureLover$, $F_2 = (\text{kevin}, \text{fluffy}) : \text{hasAnimal}$, $F_3 = (\text{kevin}, \text{tom}) : \text{hasAnimal}$, $F_4 = \text{fluffy} : \text{Cat}$, $F_5 = \text{tom} : \text{Cat}$ and $F_6 = \text{Cat} \sqsubseteq \text{Pet}$.

The KB has four worlds and the query axiom $Q = \text{kevin} : \text{NatureLover}$ is true in one of them, the one corresponding to the selection $\{(F_1,1), (F_2,1)\}$. The probability of the query is $P(Q) = 0.5 \cdot 0.6 = 0.3$.  

**Reasoning with Probabilistic Logics**
Sometimes we have to combine knowledge from multiple, untrusted sources, each one with a different reliability. Consider a KB similar to the one of Example 3.1 but where we have a single cat, fluffy.

\[\exists \text{hasAnimal.Pet} \subseteq \text{NatureLover} \quad (\text{kevin,fluffy} : \text{hasAnimal} \quad \text{Cat} \subseteq \text{Pet}\]

and there are two sources of information with different reliability that provide the information that fluffy is a cat. On one source the user has a degree of belief of 0.4, i.e., he thinks it is correct with a 40% probability, while on the other source he has a degree of belief 0.3. The user can reason on this knowledge by adding the following statements to his KB:

\[0.4 \cdot \text{fluffy : Cat} \quad 0.3 \cdot \text{fluffy : Cat}\]

The two statements represent independent evidence on fluffy being a cat. We associate \(F_1\) (\(F_2\)) to the first (second) probabilistic axiom.

The query axiom \(Q = \text{kevin : NatureLover}\) is true in 3 out of the 4 worlds, those corresponding to the selections \(\{(F_1,1),(F_2,1)\}, \{(F_1,1),(F_2,0)\}, \{(F_1,0),(F_2,1)\}\). So \(P(Q) = 0.4 \cdot 0.3 + 0.4 \cdot 0.7 + 0.6 \cdot 0.3 = 0.58\). This is reasonable if the two sources can be considered as independent. In fact, the probability comes from the disjunction of two independent Boolean random variables with probabilities respectively 0.4 and 0.3: \(P(Q) = P(X_1 \lor X_2) = P(X_1) + P(X_2) - P(X_1 \land X_2) = P(X_1) + P(X_2) - P(X_1)P(X_2) = 0.4 + 0.3 - 0.4 \cdot 0.3 = 0.58\)

4 Querying KBs

Traditionally, a reasoning algorithm decides whether an axiom is entailed or not by a KB by refutation: the axiom \(E\) is entailed if \(\neg E\) has no model in the KB. Besides deciding whether an axiom is entailed by a KB, we want to find also explanations for the axiom.

The problem of finding explanations for a query has been investigated by various authors (Schlobach and Cornet 2003; Kalyanpur et al. 2005; Kalyanpur 2006; Kalyanpur et al. 2007; Halaschek-Wiener et al. 2006). It was called axiom pinpointing in (Schlobach and Cornet 2003) and considered as a non-standard reasoning service useful for tracing derivations and debugging ontologies. In particular, in (Schlobach and Cornet 2003) the authors define minimal axiom sets (\(\text{MinA}\)s for short).

**Definition 4.1 (\(\text{MinA}\))**

Let \(K\) be a knowledge base and \(Q\) an axiom that follows from it, i.e., \(K \models Q\). We call a set \(M \subseteq K\) a minimal axiom set or \(\text{MinA}\) for \(Q\) in \(K\) if \(M \models Q\) and it is minimal w.r.t. set inclusion.

The problem of enumerating all \(\text{MinA}\)s is called \(\text{min-a-enum}\). \(\text{All-MINAs}(Q,K)\) is the set of all \(\text{MinA}\)s for query \(Q\) in knowledge base \(K\). Reasoners such as Pellet solve the \(\text{min-a-enum}\) problem by finding a single \(\text{MinA}\) using a tableau algorithm and then applying the hitting set (Reiter 1987) algorithm for finding all the others.

A tableau is a graph where each node represents an individual \(a\) and is labeled with the set of concepts \(L(a)\) it belongs to. Each edge \((a,b)\) in the graph is labeled with the set of roles to which the couple \((a,b)\) belongs. Then, a set of consistency preserving tableau expansion rules are repeatedly applied until a clash (i.e., a contradiction) is detected or a clash-free graph is found to which no more rules are applicable. A clash is for example a couple \((C,a)\) where \(C\) and \(\neg C\) are present in the label of a node, i.e. \(C, \neg C \subseteq L(a)\).

Some expansion rules are non-deterministic, i.e., they generate a finite set of tableaux. Thus the algorithm keeps a set of tableaux that is consistent if there is any tableau in
it that is consistent, i.e., that is clash-free. Each time a clash is detected in a tableau \( G \), the algorithm stops applying rules to \( G \). Once every tableau in \( T \) contains a clash or no more expansion rules can be applied to it, the algorithm terminates. If all the tableaux in the final set \( T \) contain a clash, the algorithm returns unsatisfiable as no model can be found. Otherwise, any one clash-free completion graph in \( T \) represents a possible model for the concept and the algorithm returns satisfiable. The hitting set algorithm is a black box method: it repeatedly removes an axiom from the KB and then computes again a MinA recording all the different MinAs so found.

\textsc{min-a-enum} is required to answer queries to KBs following the DISPONTE semantics. To compute the probability of a query, the explanations must be made mutually exclusive, so that the probability of each individual explanation is computed and summed with the others. This can be done by exploiting a splitting algorithm as shown in (Poole 2000). Alternatively, we can assign independent Boolean random variables to the axioms contained in the explanations and defining the Disjunctive Normal Form (DNF) Boolean formula \( f_K \) which models the set of explanations. Thus \( f_K(X) = \bigvee_{\kappa \in K} \bigwedge_{(E,i)} X_i \bigwedge_{(E,0)} \neg X_i \) where \( X = \{X_i|(E_i,k) \in \kappa, \kappa \in K\} \) is the set of Boolean random variables. We can now translate \( f_K \) to a Binary Decision Diagram (BDD), from which we can compute the probability of the query with a dynamic programming algorithm that is linear in the size of the BDD.

5 The algorithms TRILL and TRILL\(^P\)

TRILL (Zese et al. 2013) implements the tableau algorithm using Prolog. In this way, we do not have to implement a search strategy, such as the hitting set algorithm, because the management of the non-determinism is demanded to Prolog. TRILL takes as input an OWL DL ontology translated into Prolog facts by using the Thea2 library (Vassiliadis et al. 2009). For example, a subclass axiom \( \text{Cat} \sqsubseteq \text{Pet} \) is translated into \text{subClass}(‘Cat’,’Pet’) while for more complex axioms, Thea2 uses Prolog lists, so the axiom \( \text{NatureLover} \equiv \text{PetOwner} \sqcup \text{GardenOwner} \) is translated into \text{equivalentClasses}(['NatureLover',unionOf(['PetOwner','GardenOwner'])]).

TRILL builds a tableau following the tableau algorithm. The non-deterministic rules are treated differently from the deterministic ones. While the latter ones are implemented by predicates that take as input a tableau and return a single tableau, the former ones are implemented by predicates that take as input a tableau but return a list of tableaux from which one is non-deterministically chosen. The computation of \text{All-MINAS}(Q,K)\) is performed by simply calling \text{findall/3} over the tableau predicate.

A new version of TRILL, called TRILL\(^P\), resolves the axiom pinpointing problem by computing a pinpointing formula (Baader and Peñaloza 2010a; Baader and Peñaloza 2010b) instead of a set of MinAs. To define the pinpointing formula we first have to associate a Boolean variable to each axiom of the KB \( K \). The pinpointing formula is a monotone Boolean formula on these variables. This formula compactly encodes the set of all MinAs. Let assume that each axiom \( E \) of a KB \( K \) is associated with the propositional variable \( \text{var}(E) \). The set of all propositional variables is indicated with \( \text{var}(K) \). A valuation \( \nu \) of a monotone Boolean formula is the set of propositional variables that are true. For a valuation \( \nu \subseteq \text{var}(K) \), let \( \mathcal{K}_\nu := \{ t \in K | \text{var}(t) \in \nu \} \).
Definition 5.1 (Pinpointing formula)
Given a query $Q$ and a KB $\mathcal{K}$, a monotone Boolean formula $\phi$ over $\text{var}(\mathcal{K})$ is called a pinpointing formula for $Q$ if for every valuation $\nu \subset \text{var}(\mathcal{K})$ it holds that $\mathcal{K}_\nu \models Q$ if $\nu$ satisfies $\phi$.

In Lemma 2.4 of (Baader and Peñaloza 2010b), the authors proved that we can obtain all MinAs from a pinpointing formula by transforming the formula into DNF and removing disjuncts implying other disjuncts. From this formula, the construction of BDD can be performed as for MinAs. For formal definitions see (Baader and Peñaloza 2010a; Baader and Peñaloza 2010b).

Example 5.1 (Pinpointing formula)
Consider the KB of Example 3.1 with the same association between Boolean variables and axioms. Let $Q = \text{kevin : NatureLover}$ be the query, then $\text{All-MinAs}(Q, \mathcal{K}) = \{\{F_2, F_4, F_5, F_1\}, \{F_3, F_5, F_6, F_1\}\}$, while the pinpointing formula is $((F_2 \land F_4) \lor (F_3 \land F_5)) \land F_6 \land F_1$.

In order to build the BDDs and compute the associated probabilities, TRILL and TRILL$^P$ exploit a Prolog library of the cplint suite (Riguzzi 2009). The code of TRILL and TRILL$^P$ is available at https://sites.google.com/a/unife.it/ml/trill.

6 Related Work
DL reasoners written in Prolog do not need to implement a backtracking algorithm but can exploit Prolog backtracking facilities for performing the search. This has been observed in various works. Beckert and Posegga (1995) proposed a tableau reasoner in Prolog for FOL based on variable-free semantic tableaux, but it is not tailored to DLs. Meissner (2004) presented the implementation of a Prolog reasoner for the DL $\mathcal{ALCN}$. Herchenröder (2006) improved it by implementing heuristic search techniques to reduce the running time. Faizi (2011) added to its work the possibility of returning explanations for queries w.r.t. $\mathcal{ALC}$ KBs. Hustadt et al. (2008) presented the KAON2 algorithm that exploits basic superposition, a refutational theorem proving method for FOL with equality and a new inference rule, called decomposition, to reduce a $\mathcal{SHIQ}$ KB into a disjunctive datalog program. Lukácsy and Szeredi (2009) presented DLog, that is an ABox reasoning algorithm for the $\mathcal{SHIQ}$ language. It allows to store the content of the ABox externally in a database and to answer ABox queries by transforming the KB into a Prolog program. TRILL and TRILL$^P$ differ from these works for the considered DL and, in particular, from DLog for the capability of answering general queries.

Bruynooghe et al. (2010) presented FOProllog that is based on Problog, in which a program contains a set of probabilistic facts, i.e. facts annotated with probabilities, and a set of general clauses which can have positive and negative probabilistic facts in their body. Each fact is assumed to be probabilistically independent. It follows the distribution semantics and exploits BDDs to compute the probability of queries. FOProllog is a reasoner for FOL that is not tailored to DLs, so the algorithm could be suboptimal. It does not exploit a tableau algorithm and cannot manage probabilistic facts which are annotated with more than one probability value.

A different approach is the one of Ricca et al. (2009) that presented OntoDLV, a system for reasoning on logic-based ontology representation language, called OntoDLP.
OntoDLP is an extension of (disjunctive) ASP and can interoperate with OWL. OntoDLV rewrites the OWL KB into the OntoDLP language, can retrieve information directly from external OWL Ontologies and answers queries by using ASP.

BUNDLE (Riguzzi et al. 2013a) is a probabilistic reasoner that computes the probability of queries from probabilistic KBs that follow the DISPONTE semantics. It is based on Pellet and is completely written in Java. It exploits a modified version of Pellet for finding the All-MINAs set and then it translates it into a BDD from which it computes the probability of the query. Similarly to BUNDLE, PRONTO (Klinov 2008) is based on Pellet and performs inference on P-SHIQ(D) KBs in which the probabilistic part contains conditional constraints of the form \( (D|C)[l,u] \) that informally mean “generally, if an object belongs to \( C \), then it belongs to \( D \) with a probability in the interval \( [l,u] \)”. P-SHIQ(D) (Lukasiewicz 2008) uses probabilistic lexicographic entailment from probabilistic default reasoning and allows both terminological and assertional probabilistic knowledge about instances of concepts and roles. P-SHIQ(D) is based on Nilsson’s probabilistic logic (Nilsson 1986) in which the probabilistic interpretation \( Pr \) defines a probability distribution over the set of interpretations \( Int \) instead of a probability distribution over theories. The probability of a logical formula \( F \) according to \( Pr \), denoted \( Pr(F) \), is the sum of all \( Pr(I) \) such that \( I \in Int \) and \( I \models F \).

7 Experiments

We did several experiments in order to evaluate the performances of the algorithms we have implemented. Here we report a comparison between the performances of TRILL, TRILL\(^P\) and BUNDLE when computing probability for queries. We used four different knowledge bases of various complexity: 1) BRCA\(^1\) models the risk factor of breast cancer; 2) an extract of the DBPedia\(^2\) ontology obtained from Wikipedia; 3) Biopax level 3\(^3\) models metabolic pathways; 4) Vicodi\(^4\) contains information on European history.

For the tests, we used a version of the DBPedia and Biopax KBs without the ABox, a version of the BRCA with an ABox containing 1 individual and a version of Vicodi with an ABox containing 19 individuals. To each KB, we added 50 probabilistic axioms. For each datasets we randomly created 100 different queries. In particular, for the DBPedia and Biopax datasets we created 100 subclass-of queries while for the other KBs we created 80 subclass-of and 20 instance-of queries. For generating the subclass-of queries, we randomly selected two classes that are connected in the hierarchy of classes contained in the ontology, so that each query had at least one explanation. For the instance-of queries, we randomly selected an individual \( a \) and a class to which \( a \) belongs by following the hierarchy of the classes starting from the class to which \( a \) is instantiated in the KB.

Table 1 shows, for each ontology, the average number of different MINAs computed and the average time in seconds that TRILL, TRILL\(^P\) and BUNDLE took for answering the queries. In particular, the BRCA and the version of DBPedia that we used contain a large number of subclass axioms between complex concepts. These preliminary tests show that

\(^1\) http://ww2.cs.man.ac.uk/~klinovp/pronto/brc/cancer_cc.owl
\(^2\) http://dbpedia.org/
\(^3\) http://www.biopax.org/
\(^4\) http://www.vicodi.org/
both TRILL and TRILL\textsuperscript{P} performances can sometimes be better than BUNDLE, even if they lack all the optimizations that BUNDLE inherits from Pellet. This represents evidence that a Prolog implementation of a Semantic Web tableau reasoner is feasible and that may lead to a practical system. Moreover, TRILL\textsuperscript{P} presents an improvement of the execution time with respect to TRILL when more MinAs are present.

### 8 Open Issues and expected achievement

Our work aims at developing fast algorithms for performing inference over probabilistic DISPONTE semantics. Section 5 shows that TRILL and TRILL\textsuperscript{P} can compute the explanations for a query and its probability w.r.t. a SHOIN(D) and an ALC probabilistic KB respectively. For the future we plan to improve the performances of both algorithms.

We are also studying the problem of lifted inference for probabilistic logic programming using lifted variable elimination. We are adapting the Generalized Counting First Order Variable Elimination (GC-FOVE) algorithm presented in (Taghipour et al. 2013) to probabilistic logic programming under the distribution semantics. To this purpose, we are developing the system LP\textsuperscript{2} that extends GC-FOVE by introducing two new operators, heterogeneous sum and heterogeneous multiplication. This work will be presented at the ICLP 2014 main conference.

A second line of research is the problem of learning the parameters and the structure of a DISPONTE KB. Along this line, in (Riguzzi et al. 2013b) we presented a learning algorithm, called EDGE, that learns the parameters by taking as input a DL theory and a number of examples that are usually concept assertions divided into positive and negative examples. EDGE first computes, for each example, the BDD encoding its explanations, then it executes an Expectation-Maximization (EM) algorithm, in which the functions Expectation and Maximization are repeatedly applied until the log-likelihood of the examples reaches a local maximum. Moreover, we are working on extending EDGE in order to learn also the structure of a DISPONTE KB together with the parameters by adapting the CELOE algorithm (Lehmann et al. 2011).

### 9 Conclusions

In this paper we presented two algorithms TRILL and TRILL\textsuperscript{P} for reasoning on DISPONTE KBs which are written in Prolog. The experiments show that Prolog is a viable language for implementing DL reasoning algorithms and that the performances of the two presented algorithms are comparable with those of a state-of-art reasoner.
10 Acknowledgements

This work was started by the Artificial Intelligence research group of the engineering department of the University of Ferrara. We would personally thank my colleagues and friends (in alphabetical order) Elena Bellodi, Evelina Lamma and Fabrizio Riguzzi.

References


Patel-Schneider, P. F., Horrocks, I., and Bechhofer, S. 2003. Tutorial on OWL.


Shearer, R., Motik, B., and Horrocks, I. 2008. Hermit: A highly-efficient owl reasoner. In OWLED.


