A Fuzzy Datalog Deductive Database System

Pascual Julián-Iranzo, Fernando Sáenz-Pérez

Abstract—This paper describes a proposal for a deductive database system with fuzzy Datalog as its query language. Concepts supporting the fuzzy logic programming system Bousi∼Prolog are tailored to the needs of the deductive database system DES. We develop a version of fuzzy Datalog where programs and queries are compiled to the DES core Datalog language. Weak unification and weak SLD resolution are adapted for this setting, and extended to allow rules with truth degree annotations. We provide a public implementation in Prolog which is open-source, multiplatform, portable, and in-memory, featuring a graphical user interface. A distinctive feature of this system is that, unlike others, we have formally demonstrated that our implementation techniques fit the proposed operational semantics. We also study the efficiency of these implementation techniques through a series of detailed experiments. Moreover, a database example for a recommender system is used to illustrate some of the features of the system and its usefulness.

Index Terms—Deductive Database, Fuzzy Logic Programming, Fuzzy Prolog, Weak Unification, Bousi∼Prolog, Datalog Educational System.

I. INTRODUCTION

Fuzzy Logic Programming integrates concepts coming from fuzzy logic [1] into pure logic programming [2] in order to deal with imprecise information, uncertainty and/or vagueness by using declarative techniques. In the last decade there has been a renewed interest in this amalgamation as revealed in the works [3], [4]. There is not a common method for introducing fuzzy concepts into logic programming. In particular, Bousi∼Prolog [5], [6], an extension of the Prolog language, follows the approach of [7], where the syntactic unification mechanism of classical SLD resolution is replaced by a weak unification algorithm, based on proximity/similarity relations. This algorithm provides a weak most general unifier as well as a numerical value, called the approximation degree. Intuitively, the approximation degree represents the truth degree associated with the (query) computed instance. Programs written in this language consist, in essence, of a set of ordinary (Prolog) clauses jointly with a set of “proximity equations” which play an important role during the unification process.

Datalog [8] is a query language for deductive databases that can be seen as a syntactic subset of Prolog. Pure Datalog is a truly declarative language because the order neither of rules nor goals in the program do affect operational semantics (in particular, non-logic constructors are disallowed, as the cut), but it is not Turing-complete as it is meant as a database query language.

Fuzzy Datalog [9] is an extension of Datalog-like languages using lower bounds of uncertainty degrees in facts and rules. Akin proposals as [10], [11] explicitly include the computation of the rule degree, as well as an additional argument to represent this degree. However, and similar to Bousi∼Prolog, we are interested in removing this burden from the user with an automatic rule transformation that slides both the degree argument and the explicit call to fuzzy connective computations in user rules. In addition, we provide support for several proximity/similarity relations such as in Bousi∼Prolog.

Thus, in this paper, we are interested in the implementation of a deductive fuzzy database with such features by extending the Datalog Educational System (DES) [12] into a system that we call FuzzyDES from here on. The DES system is a deductive database targeted at teaching databases and their query languages. We focus on Datalog to be extended with fuzzy relations (i.e., relations where each data tuple is associated to an approximation degree), and weak unification and resolution. By contrast to a fuzzy Prolog system, answers are (multi)-sets instead of just single answers retrieved via backtracking. Whereas queries solved under SLD resolution can easily develop non-termination, queries under tabled SLD resolution [13] are ensured to terminate for user predicates, which is a natural requirement in the database arena (cf. SQL). Due to the interactive nature of DES, we keep this feature in the fuzzy setting to let users play and interact with the system by providing commands to assert new rules and facts on the fly.

For the implementation, we transfer the techniques developed for the Bousi∼Prolog system into the DES system. In general, following the approach of the high level implementation of Bousi∼Prolog, a FuzzyDES program is compiled into a set of core Datalog rules, which in turn are interpreted by a deductive engine implemented in Prolog. The translation includes calls to a collection of auxiliary predicates able to reproduce the Weak SLD resolution with graded rules (i.e., annotated rules with weights acting as truth degrees). Since Datalog differ from Prolog (e.g., non-compound terms and ground answers), we adapt and specialize the Weak SLD resolution (WSLD) procedure found in the Bousi∼Prolog system to DES. A distinctive feature of the built system is that, unlike others, we have formally demonstrated that our techniques for implementing the WSLD resolution operational semantics are correct, i.e., they produce the expected answers and only them. We also study the efficiency of these implementation techniques through a series of detailed experiments.

To the best of our knowledge, there is no a publicly-available implementation of a fuzzy Datalog system as the
one we propose in this work. This paper describes our approach to develop such a system. Our motivation lies in that, although SQL (as a query language) dominates the database panorama, over the years SQL has become quite complex and some statements have started to cause problems for end users and even for professionals. Also, one of the main advantages of deductive databases is the ability to specify recursive rules without the limitations present in SQL systems (linearity and termination control). This, jointly with the addition of fuzzy features, can facilitate the flexible use of databases and knowledge bases. Additionally, in the last years we have witnessed an increased interest in recursive Datalog queries in a variety of application domains such as data integration, information extraction, networking, program analysis, security, and even cloud computing (cf. Section III). Finally, it is important to emphasize that our system has been specially designed to facilitate flexibility for fuzzy querying and answering of questions, encapsulating fuzzy management. This is a definitive feature of our system that do not have others based on more traditional techniques, in which the user has to set certain parameters (as truth degrees and their manipulation) when specifying their queries and rules. This is an inherited characteristic of Bouși–Prolog that allows a complete separation when specifying logic, vague knowledge and control. All of these are appreciable features that facilitate the modelling of knowledge systems from a more declarative perspective.

The following section gives a practical motivation to this work by describing an application example of FuzzyDES showing its ability to model uncertain knowledge and facilitate flexible query answering.1

II. A MOTIVATING EXAMPLE

Recommender systems are an effective way to assist people by offering advice on finding suitable products and services to facilitate online decision-making [14]. Subjective information (such as how a person evaluates a product or service, and how another person trust an opinion depending on its confidence) can be specified with linguistic, fuzzy information. In order to motivate our work, as an example of a fuzzy database in DES, we apply these ideas to modelling a small recommender system for restaurants in Madrid. We consider people interested in asking questions relating the location of the restaurant with respect to its own location, the quality of the restaurant in terms of other’s opinions, and the type of food served. Thus, we can distinguish two fuzzy relations: A proximity relation near (reflexive and symmetric) for representing walking distances, and a predefined similarity relation ~ (which is, in addition, transitive) for representing quality degrees. These are defined in the next code excerpt of the database file that can be downloaded from http://des.sourceforge.net/fuzzy/recommender.dl.

}\$fuzzy_relation\left(near,\left[\text{reflexive},\text{symmetric}\right]\right)\).\n\begin{align*}
\text{sol near callao} &= 0.6. \\
\text{sol near cruz} &= 0.5. \\
\text{callao near plaza}\_\text{espa\~na} &= 0.4. \\
\end{align*}\n\$fuzzy_relation\left(~,\left[\text{reflexive},\text{symmetric},\text{transitive}\right]\right)\).\n
1The documentation in the downloadable system contains some other application examples and all the available commands.

The first assertion fuzzy_relation defines the operator near as a proximity relation for a walking distance with the reflexive and symmetric properties. Analogously, a similarity relation ~ is explicitly defined next (in fact, its assertion can be removed since it is defined as such by default). Proximity equations, as plain\textquoteleft\textquoteleft good\textquoteleft\textquoteleft0.5, specify the degree of semantic similarity for two different syntactic symbols. In this particular case it states that plain and good are similar with approximation degree 0.5.

A fuzzy relation confidence1 defines the degree of reliability (in the opinion) which a given type of user deserves. For example, a local guide is assumed to be a person including serious comments in the database where normal and casual users may not, and with different confidences:

\begin{align*}
\text{confidence\left(local\_guide\right)} & \text{ with 0.9.} \\
\text{confidence\left(normal\_user\right)} & \text{ with 0.5.} \\
\text{confidence\left(casual\_user\right)} & \text{ with 0.3.} \\
\end{align*}

Next, several relations are defined with facts in the database:

The relation restaurant/3 relates the name of a restaurant, its location, and food served. Users and their types are related in the relation user/3. Finally, user comments are represented in the relation comment/3 relating user, restaurant, and comment.

\begin{align*}
\text{restaurant\left(don\_oso, cruz, burguer\right).} \\
\text{restaurant\left(roque, sol, rice\right).} \\
\text{restaurant\left(tagliatella, benavente, italian\right).} \\
\text{user\left(juan, local\_guide\right).} \\
\text{user\left(sara, normal\_user\right).} \\
\text{user\left(pepe, casual\_user\right).} \\
\end{align*}

\begin{align*}
\text{comment\left(juan, don\_oso, plain\right).} \\
\text{comment\left(juan, rodilla, good\right).} \\
\text{comment\left(pepe, roque, excellent\right).} \\
\text{comment\left(sara, tagliatella, very\_good\right).} \\
\end{align*}

A recommendation relies on the quality of a restaurant, which is defined with a single rule that takes into account the comment a user has provided, the type of this user, and its confidence:

\begin{align*}
\text{quality\left(Restaurant, Quality\right)} & \text{ :: comment\left(User, Restaurant, Quality\right),} \\
& \text{ user\left(User, Type\right), confidence\left(Type\right).} \\
\end{align*}

So, in order to provide recommendations, the rule recommend relates restaurants (Restaurant) with the location of the user (Origin), serving certain food (Food) with an acknowledged quality (Quality):

\begin{align*}
\text{recommend\left(Origin, Food, Quality, Restaurant\right)} & \text{ :: restaurant\left(Restaurant, Location, Food\right),} \\
& \text{ Location near Origin, quality\left(Restaurant, Quality\right).} \\
\end{align*}

Contrary to Fuzzy Prolog implementations, the set-oriented approach of deductive databases makes queries to return all the answers, similar to relational databases, so that a query to this database can return several recommendations with different approximation degrees at once. For instance, the query in the following system session returns recommendations for a user located at sol:

\begin{verbatim}
DES> /system_mode fuzzy
FDES> /consult recommender
Info: 22 rules consulted.
\end{verbatim}
FDES> recommend(sol,Food,Quality,Restaurant)
Info: Processing:
answer(Food,Quality,Restaurant) :-
  recommend(sol,Food,Quality,Restaurant).
  { answer(snacks,good,rodilla)with 0.6,
    answer(burguer,plain,don_os0)with 0.5,
    answer(rice,excellent,roque)with 0.3 }

In this session, after switching the deductive system to the fuzzy setting, the command consult loads the database located in the given file (with default extension .dl). The query returns all possible restaurants indicating the type of food served, quality and the recommended restaurant. Each answer tuple is ordered by default with respect to descending approximation degrees. The first tuple indicates that rodilla has a support degree of 0.6, which is a value constructed by taking into account that the restaurant is located at callao, which is near sol. Also, there is a comment from the local guide juan (with a support of 0.9) stating that the restaurant is good. The next tuple in the answer refer to the restaurant don_os0, receiving a support degree of 0.5 with quality plain, because the same local guide commented on this restaurant with this quality level for a restaurant located at cruz, near sol. The last tuple receives a small support because the comment was raised by a user whose type is scored rather low.

We can interactively add information about the type of the food served with:
FDES> /assert burguer~fast_food=0.7.
FDES> /assert snacks~fast_food=0.9.

Then, a similar query but looking for good fast food near sol (with no need to reload the database) would retrieve:
FDES> recommend(sol,fast_food,good,Restaurant)
Info: Processing:
answer(Restaurant) :-
  recommend(sol,fast_food,good,Restaurant).
  { answer(rodilla)with 0.6, answer(don_os0)with 0.5 }

There are really two tuples for don_os0 fulfilling the question, with the same support degree (one for plain and other for good quality). Due to the default set-oriented behavior of the system, duplicates are removed. However, duplicates can be enabled with the command /duplicates on, and the same query would return 3 tuples.2

An approximation degree threshold (λ-cut) can be stated with the command /lambda_cut Value, which prunes computations as soon as a degree greater than the threshold is computed as a result of an application of the t-norm Δ, which is a binary truth function generalizing classical conjunction. In this last example, the second answer would be removed from the answer for a λ-cut of 0.55.

Facts and rules can also be interactively added (with the command /assert Rule or from a file with /reconsult File) so that the database have not to be recompiled each time it is modified. Finally, a query for the predicate recommend with variables in all its arguments would return all possible recommendations. (The reader is encouraged to try the system with different queries.)

2Nonetheless, as it will explained later in Subsection IV-F, enabling answer subsumption automatically removes output tuples with equal or lesser approximation degrees.

III. DATALOG EDUCATIONAL SYSTEM

The Datalog Educational System (DES) [12] is a free, open-source, multiplatform, portable, in-memory, Prolog-based implementation of a deductive database system. DES 5.0.1 (des.sourceforge.net) is the current release, which includes several query languages: Datalog, SQL, Relational Algebra and Relational Calculi. Fuzzy Datalog is the main addition for this new major release. DES features tabling, types, integrity constraints, stratified negation [8], persistency, full-fledged arithmetic, ODBC connections to relational databases, novel approaches to Datalog and SQL declarative debugging [15], [16], test case generation for SQL views [17], null value support, outer join, and aggregate predicates and functions [12]. This system is used world-wide in many universities (e.g. Imperial College London –UK–, TU Munchen –Germany–, Université Lille 1 –France–, UCLA –USA– or The University of Sydney –Australia–) for teaching deductive databases (See http://des.sourceforge.net/html/what_s_des_for_html, in its web page, for an extensive list of institutions), and used as a test-bed for research (Mozilla leaks, smart grid networks, deductive data warehouses, declarative debugging, . . .), and featuring more than 71K downloads up to now.

A. Syntax

A deductive database in DES is defined by a normal logic program with rules of the form $A \leftarrow Q$, where the body $Q \equiv L_1 \land \ldots \land L_n$ is a query, the head $A$ is an atom and, each $L_i$ is a literal. In this paper, we restrict ourselves to definite clauses (with positive literals). Thus, a literal can be a call to either a defined predicate or to a built-in predicate. A disjunctive rule $A \leftarrow Q_1 \lor Q_2$ is understood as the rules $A \leftarrow Q_1$ and $A \leftarrow Q_2$. Textual syntax in the system follows Prolog convention: Variables start with either upper-case or an underscore, and predicate symbols and string constants either start with down-case or are delimited by single quotes. The left implication $\leftarrow$ is written as :-, the conjunction with a comma (,), and the disjunction with a semicolon (;). We interchangeably use the terms: predicate - relation, clause - rule, and goal - query.

B. Safe Databases

Though the Datalog language might be understood as a syntactic subset of Prolog, it is claimed to be a true declarative language (see, e.g., [18]) in the sense that rule and goal ordering in a program does not affect declarative semantics. However, as a database language, users expect termination and finite answers, so that some restrictions to programs are imposed. The first restriction is about safe rules [8], [19], that ensure closed (ground) answers. Open answers (i.e., including unbound variables) are not allowed as they can represent infinite tuples (for example, the answer $p(X)$ represents $\forall X p(X)$, a domain which is not defined as finite in the database). A second restriction refers to built-ins because whereas user-defined predicates represent finite relations, some useful built-ins represent infinite relations. For example, a comparison operator $(\subseteq, >, \ldots)$ relates two arguments representing a true relation for all the possible elements fulfilling the comparison

---

**Facts and rules can also be interactively added (with the command /assert Rule or from a file with /reconsult File) so that the database have not to be recompiled each time it is modified.**
(e.g., \(X < Y\) represents all the pairs \((X, Y)\) such that \(X\) is less than \(Y\)). For ensuring a finite answer, these operators demand ground arguments.

C. Stratification and Aggregates

Deductive system implementations require some form of fixpoint computation to deduce the meaning of programs and queries. Keeping this computation monotonic is achieved by the well-known technique called stratification [8], which is a syntactic restriction that rejects programs which combine both negation and recursion in a computation path. Though in our work we do not deal with negation, we still take advantage of this technique for computing aggregates. Computing an aggregate can be a source of non-monotonicity because in a given iteration of a fixpoint computation, all the tuples for the relation on which the aggregate operates are not available in general. So, there exists the risk of deducing an incorrect outcome in a given iteration that can be neglected in a further iteration. For instance, we can assume that the tuple \(r(0)\) is in the current fixpoint iteration, so that computing the maximum of the single argument of \(r\) yields 0. But it is possible that in a next iteration, a new tuple (say \(r(1)\)) is computed, therefore neglecting the previous computed value for the aggregate. Thus, stratification is applied to ensure that all the tuples of the relation \(r\) are computed before trying to compute the aggregate.

A stratification for a program is built with the aid of a predicate dependency graph (PDG) [19], showing both the positive and negative dependencies between predicates in the program. Each node in this graph is a program predicate symbol, and there are as many nodes as such symbols in the program. Arcs come from each predicate in a rule body (antecedent) to its rule predicate (consequent). If the antecedent occurs as an aggregate argument, its arc is labelled as negative, and positive otherwise. A stratification collects predicates into numbered strata so that, given the function \(\text{str}(\Pi, p)\) which assigns a stratum number to predicate \(p\) in a database \(\Pi\), then for a positive arc \(p \rightarrow q\), \(\text{str}(\Pi, p) \leq \text{str}(\Pi, q)\), and for a negative arc \(p \leftarrow q\), \(\text{str}(\Pi, p) < \text{str}(\Pi, q)\).

In this paper, we use the aggregate \(\text{group_by}(R, Vs, E)\), where \(R\) is a relation call, \(Vs\) are the variables in \(R\) used as a grouping criterion, and \(E\) is a Boolean expression including in general an aggregate. For example, the query \(\text{group_by}(r(X, Y), [X], \text{max}(Y))\) is intended to compute the maximum value of \(Y\) for each group formed by tuples with the same value for \(X\). Thus, if the meaning of \(r\) is defined by \{ \(r(a, 0)\), \(r(a, 1)\), \(r(b, 3)\) \}, then the previous query returns \{ \(\text{answer}(a, 1)\), \(\text{answer}(b, 3)\) \}.

D. Tabling

Instead of inferring the meaning of the whole database, DES solves queries with a top-down-driven, bottom-up fixpoint computation based on tabling, following the ideas found in [20]. In this section we assume safe databases.

Implementing tabling in DES resorts to a call table \(ct\) which stores the goal calls made along resolution as atom entries, \(\phi\), and answers in an answer table \(at\) as \(id : \psi\) entries, where \(id\) is a clause identifier and \(\psi\) is a positive ground atom. Filling answer and call tables is due to the memo function which proceeds by tabled SLD resolution [13], [20], [21]. Along resolution, a call to a goal is executed if there are no entry in the call table subsuming the current call. If so, this call is added to the call table. Otherwise, it means that the results for a more general call has been performed and its answers are available in the answer table, which can be simply retrieved for the current call. Analogously, upon resolving a given call, its result is added as an entry to the answer table if there is no other entry in the same table subsuming it. While in logic programming systems a subsumption test involves open entries in general, in deductive systems as DES, a subsumption test cannot involve open entries because safety. Thus, in this latter case, answer subsumption simply leads to a membership test.

Filling the answer and call tables is done by strata by ensuring that the meaning of atoms which are required to prove other goals are already in the answer table. Such atoms can be either those in negative literals (which are out of the scope of the current work) or those in aggregates. So, following the stratification for the database \(\Pi\), for a given goal \(\phi\), a goal dependency graph (\(\text{gdg}(\Pi, \phi)\)) is computed, which is the subgraph of the PDG such that contains all reachable nodes from \(\phi\) in \(\Pi\). Then, for each node \(p_i\) in the subgraph such that there is a negative arc coming out from \(p_i\), an open goal \(\phi_i\) is built with the same arity as \(p_i\). Goals \(\phi_i\) are ordered by \(\text{str}(\Pi, \phi_i)\), so that lower-strata goals will be computed before upper-strata goals.

The stratified meaning of a program restricted to a goal is got by filling the tables as specified next:

\[
\text{Definition 3.1: Stratified Meaning of a Program restricted to a Goal. Given a program } \Pi \text{ and a goal } \phi_k \text{ for } n \geq 0,
\]

\[
< ct_i, at_i > = \bigcup_{n \geq 0} \text{memo}^n(\phi_i, \Pi, ct_{i-1}, at_{i-1})
\]

where \(\text{gdg}(\Pi, \phi_k) = < N, A >, p_i \in N, i \in \{1, \ldots, k\}, q \leftarrow p_i \in A \) for some \(q, \phi_i = p_i(X_1, \ldots, X_{\text{arity}(p_i)}), \) \(X_j\) fresh variables, \(\text{arity}(p_i)\) is the arity of the predicate \(p_i\), and indexes \(i\) are ordered such that \(\text{str}(\Pi, \phi_i) \leq \text{str}(\Pi, \phi_{i+1})\), and \(\bigcup_{n \geq 0}\) is the least upper bound of the successive applications of the function \(\text{memo}\) which solves a goal using call and answer tables [13].

So, the meaning of a tabled goal is defined analogously to the meaning of a goal:

\[
\text{Definition 3.2: Meaning of a Tabled Goal. The meaning of a tabled goal } \phi \text{ w.r.t. an answer table } at \text{ for a source tuple identifier } id \text{ is defined as } \text{solve}(\phi, at) = \{ \psi \text{ such that } id : \psi \in at, \text{ and } \phi \theta = \psi \} \text{ where solve returns a bag, and } \theta \text{ is a substitution.}
\]

Thanks to tabling, while some Prolog queries (goals) are non-terminating, equivalent Datalog queries are terminating (modulo infinite built-in predicates). For example, the query \(p(X)\) in the context of the program \{ \(p(1), p(X) : \neg p(X)\) \} has the answer \{ \(p(1)\) \}. Even more, termination in Prolog might depend on rule ordering. For example, the same query \(p(X)\) in

\[3\text{An atom } A \text{ subsumes another } B \text{ if there exists a substitution } \sigma \text{ such that } A \sigma = B. \text{ That is, if } B \text{ is an instance of } A.\]
the context of the program \{ p(X); -p(Y), Y > 0, X \neq Y-1, p(1) \} do not terminate for SLD resolution as the first rule is selected infinitely many times without possibility of selecting the second one. By contrast, tabling provides the answer \{ p(0), p(1) \}.

Tabling has been acknowledged as an efficient approach in logic programming, where complexity results are given (see, e.g., [22]), so that our approach seems adequate for supporting a fuzzy deductive database system.

IV. FuzzyDES

FuzzyDES is an extension of the Datalog language with truth degree annotations and an operational semantics, that we called Weak SLD resolution, based on similarity relations. In this section, we briefly summarize the main theoretical concepts supporting FuzzyDES and we give the clues of its implementation.

A. Fuzzy Relations and Weak Unification

A binary fuzzy relation on a set \( U \ast U \) is a fuzzy subset on \( U \times U \) (that is, a mapping \( U \times U \rightarrow [0,1] \)). There are some important properties that fuzzy relations may have: (1) (Reflexive) \( R(x,x) = 1 \) for any \( x \in U \); (2) (Symmetric) \( R(x,y) = R(y,x) \) for any \( x,y \in U \); (3) (Transitive) \( R(x,y) \geq R(x,z)R(y,z) \) for any \( x,y,z \in U \); where the operator \( \triangle \) is an arbitrary t-norm. The notion of transitivity above is \( \triangle \)-transitive, and if the operator \( \triangle \) is the minimum \( \triangle \)-transitive, we speak of min-transitive.

A proximity relation is a binary fuzzy relation which is reflexive and symmetric. A proximity relation which in addition fulfills the transitivity property is called a similarity relation.

We are primarily interested in similarity relations on the alphabet of a first order language. The reason is to break the usual constraint in programming languages by which distinct different syntactic symbols represent distinct information. So, in this context, two different constant symbols or predicate/function symbols, with the same arity, can be treated as equal up to a certain degree. Syntactically equal variables are considered similar with approximation degree 1; otherwise, their approximation degree is 0.

For a Datalog language, where a term can only be either a constant or a variable (i.e., no function symbols are allowed), the similarity relation \( R \) on the symbols of the alphabet can be extended to atomic formulas as follows: Let \( p \) and \( q \) be two \( n \)-ary predicate symbols and let \( t_1, \ldots, t_n, s_1, \ldots, s_n \) be either constants or variables, then,

\[
R(p(t_1, \ldots, t_n), q(s_1, \ldots, s_n)) = R(p,q)\triangle(\triangle_{i=1}^{n} R(t_i, s_i)).
\]

FuzzyDES inherits the weak unification algorithm of Bousi~Prolog. When we work with similarity relations, this algorithm coincides with the one defined by M. Sessa [7]. Here we give our slightly different version of it, which is thresholded by a cut value \( \lambda \in [0,1] \), also known as \( \lambda \)-cut.

In the context of a similarity relation it is possible to define a weak notion of most general unifier. Let \( R \) be a similarity relation, \( \lambda \) be a cut value and \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) be two expressions. The substitution \( \theta \) is a weak unifier of level \( \lambda \) for \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) with respect to \( R \) (or \( \lambda \)-unifier) if its unification degree \( R([\mathcal{E}_1\theta],[\mathcal{E}_2\theta]) \geq \lambda \). A substitution \( \theta \) is a weak most general unifier (wmgu) of level \( \lambda \) or \( \lambda \)-wmgu, w.r.t. \( R \), for \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \), denoted by \( \text{wmgu}_R(\mathcal{E}_1, \mathcal{E}_2) \), if: (1) \( \theta \) is a \( \lambda \)-unifier of \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \); and (2) for any \( \lambda \)-unifier \( \sigma \) of \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \), the substitution \( \theta \) is more general than the substitution \( \sigma \) with level \( \lambda \); that is, there exists a substitution \( \delta \) such that, for any variable \( x \) in \( \text{Dom}(\sigma) \cup \text{Dom}(\theta) \), \( R(x\sigma, x\theta \delta) \geq \lambda \).

Weak most general unifiers are computed by means of a weak unification algorithm which is formalized as a transition system supported by a similarity-based unification relation \( \Rightarrow \). For a similarity relation \( R \) and a cut value \( \lambda \), the unification of the expressions \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) is obtained by a state transformation sequence starting from an initial state \( G_0 \equiv \{ \mathcal{E}_1 \equiv \mathcal{E}_2, id, \alpha_0 \} \), where \( id \) is the identity substitution and \( \alpha_0 = 1 : G_0, id, \alpha_0 \Rightarrow^+ (G_n, \theta_n, \alpha_n) \). When \( G_n = \emptyset \) and \( \alpha_n \geq \lambda \), the expressions \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) are unifiable by similarity with wmgua \( \theta_n \) and unification degree \( \alpha_n \). Otherwise, if \( G_n = \text{Fail}, \mathcal{E}_1 \) and \( \mathcal{E}_2 \) fail to unify.

The similarity-based unification relation \( \Rightarrow \) is defined as in the classical unification algorithm, except for the term decomposition and failure rules (see later in Subsection IV-C and in [6] for a more formal and extensive discussion of this topic). Note that, unlike [7], the resulting approximation degree is limited by a cut value \( \lambda \geq 0 \). Nevertheless, the weak unification theorem proved in [7, pag. 412] is valid in our framework.

B. Implementing Fuzzy Relations

As in Bousi~Prolog, a fuzzy relation \( R \) can be specified by stating, first, a set of what we called relationship equations and, second, its properties. A relationship equation is a program declaration with textual format \( x/a \equiv y/b = a \), which represents entries \( R(x,y) = \alpha \) of the fuzzy binary relation \( R \) (textually represented by an infix operator \( \ast \)), and its intuitive reading is that two \( n \)-ary symbols, \( x \) (\( x \)) and \( y \) (\( y \)), are related with a certain degree \( \alpha \) (\( \alpha \)), where the arity specification for constants (\( /0 \)) is omitted in the equation. Note that when the relation partially specified by the relationship equations is a proximity (or a similarity) we will often speak of proximity equations. The relationship equations specifying the standard similarity relation are represented internally by atoms \( \ast (x,y,D) \) meaning that two symbols \( x \) and \( y \) are related with the approximation degree \( D \).

The properties attached to a relation \( R \) represent its intensional description, so that users are not expected to include all the relationship equations fulfilling such properties. For instance, obtaining the (additional) entries in a similarity relation \( R \) (which are needed for the compilation of e-clauses—see Subsection IV-E–) is performed by automatically computing its \( \triangle \)-closure [23], [24]. This closure can be understood as the shortest paths in a weighted directed graph (cf. Floyd-Warshall algorithm), where the length of a path \( x_1 \overset{\alpha_1}{\rightarrow} x_2 \overset{\alpha_2}{\rightarrow} \cdots \overset{\alpha_{n-1}}{\rightarrow} x_n \) is \( \triangle_{i=1}^{n-1} \alpha_i \). By taking advantage of the DES deductive engine, this closure can be neatly specified with Datalog rules for each of the properties with no need to resorting to a specific implementation of a \( \triangle \)-closure algorithm.
In the concrete implementation of FuzzyDES, a default similarity relation $R$ is denoted by the identifier $\sim$ and it is represented internally by a set of atoms $(X,Y,D)$ relating two symbols $X$ and $Y$ with the approximation degree $D$. The similarity relation $\sim$ is intensionally defined by its properties with the following Datalog rules, where underscored variables are non-relevant variables for the outcome:

$$\sim(X,1.0) \leftarrow \sim(X,_,D1); \sim(,Y,X,_,D2).$$
$$\sim(X,Y,D) \leftarrow \sim(Y,X,D).$$
$$\sim(X,Y,D) \leftarrow \sim(X,Z,D1), \sim(Z,Y,D2), \sim(\sim,X,D=max(D1)).$$

where the call to the predicate $\sim_{t\_norm}/3$ represents the application of the t-norm $\triangle$ associated to a relation $R$ (e.g., the relation $\sim$ has associated the minimum t-norm by default), provided in its first argument, to the list of approximation degrees given in the second argument. The result of this operation is returned in the third argument of $\sim_{t\_norm}/3$. The call to the aggregate metapredicate $\text{group\_by}/3$ groups tuples in $\sim$ by the criteria $[X,Y]$, and applies the aggregate expression $D=max(D1)$ over the partitioned relation (cf. extended relational algebra operation [25]). Note that $\sim'/2$ in DES does not just represent unification, but performs expression evaluation of both arguments followed by their unification (compound terms are not allowed as data). In plain words, this operation selects among the different $\sim(X,Y,D1)$ with the same arguments $X$ and $Y$, the one which has maximum $D1$.

Solving the aggregate in the metapredicate $\sim_{t\_norm}/3$ requires that the relation $\sim$, in its first argument, must be placed in a lower stratum (similarly to what happens with the negation) than the one of the predicate $\sim$ containing the rule with the metapredicate. For a relation with no transitive property, its closure coincides with the classical closure, and the last rule simply becomes: $\sim(X,Y,D) \leftarrow \sim(X,Y,D)$, so that both $\sim$ and $\sim$ are in the same stratum. Finally, note that only some built-ins as this grouping can contain compound terms and goals as arguments.

The system includes several commands for stating the properties and t-norm of a given relation. In particular, the command /fuzzy\_relation Relation List\_Of\_Properties sets the relation name with its properties given as a list (including reflexive, symmetric and transitive). The command /t\_norm Relation TNorm sets the t-norm (goedel, lukasiewicz, product, hamacher, nilpotent), where min is synonymous for goedel, and luka for lukasiewicz) for the given relation.

C. Implementing Weak Unification

Weak unification applies both to terms and predicates. The specific weak unification algorithm is implemented following closely Martelli and Montanari’s unification algorithm for syntactic unification [26]. The occurs check is not needed in the deductive setting because compound terms are not allowed (as $X=f(X)$ in Prolog, which would represent an infinite structure).

$$\text{weak\_unify(Atomic1, Atomic2, Lambda, Degree) :- atomic(Atomic1), atomic(Atomic2), !, \text{unification\_degree(Atomic1,Atomic2,Degree)}, Degree =\geq Lambda}.$$  
$$\text{weak\_unify(Term, Variable, _Lambda, 1.0) :- nonvar(Term), var(Variable), !, Variable = Term}.$$  
$$\text{weak\_unify(Variable, Term, _Lambda, 1.0) :- var(Variable), Variable = Term}.$$  

The first clause uses the approximation degree between two constants defined by $\sim$ for returning the $\text{wmgu}$ class. The predicate $\text{unification\_degree(Atomic1, Atomic2, Degree)}$ returns this degree, and it is checked next to be above the $\lambda$-cut value. The predicate is implemented as:

$$\text{unification\_degree(Atomic,Atomic,1.0) :- (fuzzy\_relation\_property(\sim, reflexive) \rightarrow !; fail).}$$

Its first clause returns the top approximation degree for unifying a constant with itself if the relation is reflexive. Otherwise, it either returns the degree between them (if exists) or fails. The final cut removes alternatives and provides a way to select the representative of the $\text{wmgu}$ class. Hence, this implementation provides a weak most general unifier as well as a numerical value, called the unification degree in [7]. Intuitively, the unification degree will represent the truth degree associated with the (query) computed instance.

FuzzyDES implements a weak unification operator, denoted by $\sim'$, which is the fuzzy counterpart of the syntactical unification operator $\alpha$ of standard Prolog. It can be used, first, to unify two terms as in the goal $\text{Term1} \sim' \text{Term2}$, and, second, to construct expressions. So, the expression $\text{Term}1 \sim' \text{Term2}$ returns the unification degree when evaluated, and can be used wherever an expression is expected in a goal, as in $1-a\sim b>0.5$ (if $a\sim b=0.4$, then the goal succeeds because $1-0.4>0.5$).

Solving a goal $\text{Expr1 Op Expr2}$ in DES, where $\text{Op}$ is a comparison operator ($\sim$, $\sim'$, $\geq$, $\leq$, $\sim'>$, $\sim'<$, $\sim=\sim$) and each $\text{Expr}$ is an expression, amounts to evaluate both expressions and comparing them with respect to the operator (with the exception of $\sim$, which performs classical unification on evaluated terms).

D. FuzzyDES Programs and Weak SLD Resolution

FuzzyDES defines a program $\Pi$ as a fuzzy theory, that is, as a mapping applying a finite set of formulas, namely rules, into the elements (truth values) of the lattice $[0,1]$. Informally, a FuzzyDES program can be seen as a set of pairs $\langle R;\alpha \rangle$, where $R$ is a rule and $\alpha=\Pi(R)$ is a truth degree expressing the confidence which the system user has in the truth of the rule $R$. We call such a pair a graded rule. For rules, FuzzyDES follows the same syntactical conventions as described in Subsection III-A, and textual syntax of a graded rule $\langle R;\alpha \rangle$ is the textual syntax for $R$ followed by with $\alpha$.

This clause is introduced for efficiency reasons, by omitting the search for entries $\sim(\text{Atomic,Atomic,1.0})$ in the $\Delta$-closure. These extensional entries could be omitted as well.
The next definition, which formalizes FuzzyDES operational semantics, enhances the definition in Subsection 5.2 of [6] to deal with graded rules.

**Definition 4.1:** Let Π be a FuzzyDES program, R a similarity relation on the first order alphabet induced by Π, A the fixed t-norm associated to R, and λ a λ-cut. We define Weak SLD (WSLD) resolution as a transition system $\langle E, \Rightarrow_{WSLD} \rangle$ where E is a set of triples $\langle G, \theta, \alpha \rangle$ (goal, substitution, approximation degree), that we call the state of a computation, and whose transition relation $\Rightarrow_{WSLD} \subseteq (E \times E)$ is the smallest relation which satisfies:

\[
\langle (\sim A \land Q'), \theta, \alpha \rangle \Rightarrow_{WSLD} \langle (Q \land Q'), \theta, \sigma, \delta \triangleright \delta \triangleright \sigma \rangle
\]

if $\langle R \equiv (A \leftarrow Q); \delta \rangle \ll_\Pi \Pi$, \(\delta \geq \lambda\), \(\sigma = \text{wmg}(A, A')\) \neq \text{fail}, \(\delta = R(A, A, A') \geq \lambda\), and $(\delta \triangleright \delta \triangleright \lambda) \geq \lambda$.

where, Q and $Q'$ are conjuctions of atoms and the notation $R \ll_\Pi \Pi$ represents that R is a standardized apart rule in Π.

A WSLD derivation for $\Pi \cup \{G_0\}$ is a sequence of WSLD resolution steps $\langle G_0, \text{id}, 1 \rangle \Rightarrow_{WSLD} \langle G_1, \theta_1, \alpha_1 \rangle \Rightarrow_{WSLD} \ldots \Rightarrow_{WSLD} \langle G_n, \theta_n, \alpha_n \rangle$. And a WSLD refutation is a WSLD derivation $\langle G_0, \text{id}, 1 \rangle \Rightarrow_{WSLD} \langle \emptyset, \sigma, \alpha \rangle$, where $\emptyset$ is the empty clause, $(\sigma, \alpha)$ is a fuzzy computed answer, where $\sigma$ is a computed substitution and $\alpha$ its approximation degree. Certainly, a WSLD-refutation computes a family of answers, in the sense that, if $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ is a computed substitution, with degree $\alpha$, then any substitution $\theta' = \{x_1/s_1, \ldots, x_n/s_n\}$, satisfying $R(s_i, t_i) \geq \lambda$, for any $1 \leq i \leq n$, is also a computed substitution if its approximation degree $\beta = \alpha \Delta_{\Pi}(R(s_i, t_i)) \geq \lambda$.

**E. Compilation of FuzzyDES Programs**

Similarly to Bousi-Prolog, FuzzyDES implements the operational mechanism of WSLD resolution by compiling programs into core Datalog rules (instead of to Prolog clauses).

The idea is to obtain a set of Datalog rules that allows the emulation of the WSLD resolution procedure, splitting it in two steps: i) the crisp unification of a defined predicate (in the head of the transformed rule); and ii) the weak unification of the attributes of the defined predicate (in the body of the transformed rules) with the help of the built-in weak unification predicate explained in Subsection IV-C. To this end, the transformation moves each head argument to an explicit weak unification performed by a specific predicate, as formalized later.

Moreover, if there exists $R(p, q) = \alpha$ between predicates p and q, then simulating a flexible matching of these predicate symbols by using a classical unification technique can be done by introducing a new clause for each predicate q which is close to p.

The following definition formalizes the program transformation just outlined above, but first we need to introduce an extended language obtained by adding to the source language alphabet the elements of the lattice $[0,1]$ (of approximation degrees). Clauses in this extended language contain bodies with literals which are interpreted as approximation degrees. We call these clauses e-clauses (expanded clauses). Also, e-clauses with an empty head are called e-goals.

**Definition 4.2:** Let Π be a logic program, R a similarity relation on the syntactic domain generated by Π, A the fixed t-norm associated to R, and $\lambda \in [0,1]$ a cut value. Let $\langle p(1), \ldots, p(n) \leftarrow Q; \delta \rangle$ be a graded clause in Π defining the n-ary predicate p. Then, for each $R(p, q) = \alpha > \lambda$ add to the transformed program Π’ the new e-clause:

\[
q(x_1, \ldots, x_n) \leftarrow (\delta \triangleright \delta) \land x_1 \approx t_1 \land \cdots \land x_n \approx t_n \land Q.
\]

where each $x_i$ is a fresh variable and $x_i \approx t_i$ is an expression that returns the approximation degree obtained from the weak unification of the term bound to the variable $x_i$ and the term $t_i$.

Observe that, since $R(p, p) = 1$ for any symbol p, if $\langle p(t_1, \ldots, t_n) \leftarrow Q; \delta \rangle$ is in the original program, the e-clause $q(x_1, \ldots, x_n) \leftarrow \delta \land x_1 \approx t_1 \land \cdots \land x_n \approx t_n \land Q$ will be in the transformed program. Thus, we give a uniform treatment for all clauses in the transformed program. Since this transformation was initially implemented in Bousi-Prolog, we name it (for short) BPL expansion, and the transformed programs generated by its application BPL expanded programs.

This transformation has been adapted and implemented in the FuzzyDES system, where the predicate expand_rule can be executed by the FuzzyDES system. The behaviour of this predicate is illustrated by the following example.

**Example 1:** Consider the following fragment of a simple FuzzyDES program, where the truth degree of a graded rule is specified with the operator with:

\[
\text{\% PROXIMITY EQUATIONS}
\]

\[
p/1 \cdot q/i=0.0.\]

\[
\text{\% FACTS AND RULES}
\]

\[
r(a).\quad p(b).\quad p(X) :- r(X) \text{ with } 0.8.\quad q(c).
\]

After the execution of the predicate expand_rule, the following Datalog code is generated (having issued the command /fuzzy Expansion bpl):

\[
r(A, D) :- \$uni\_\text{f arguments'}([[A,a,B]]), \$t\_\text{norm'}([-\_B,D,B,D]).
\]

\[
p(A, D) :- \$uni\_\text{f arguments'}([[A,b,B]]), \$t\_\text{norm'}([-\_B,D,B,D]).
\]

\[
p(A, D) :- \$uni\_\text{f arguments'}([[A,X,B]], r(X,C)), \$t\_\text{norm'}([-\_D,O,B,C,B,D]).
\]

\[
p(A, D) :- \$over\_\text{lambda cut'}(0.6), \$uni\_\text{f arguments'}([[A,c,B]]), \$t\_\text{norm'}([-\_D,O,B,D,D]).
\]

\[
q(A, D) :- \$uni\_\text{f arguments'}([[A,c,B]]), \$t\_\text{norm'}([-\_D,O,B,D,D]).
\]

\[
q(A, D) :- \$over\_\text{lambda cut'}(0.6), \$uni\_\text{f arguments'}([[A,X,B]], r(X,C)), \$t\_\text{norm'}([-\_D,O,B,C,B,D,D]).
\]

Note that each transformed predicate has an additional argument in order to store and allow the propagation of truth degrees. Here, $\$uni\_\text{f arguments'} implements the weak unification operator $\approx$ in Definition 4.2, and $\$t\_\text{norm'} (as explained in Subsection IV-B) is an internal predicate used to propagate the truth degrees of the rules and the approximation degrees coming from the fuzzy relations. The predicate

\[\text{\% The rules listed as a result of the transformation correspond to core DATALOG, and they can be examined with the command /listing and enabling development listings with /development on.}\]
Fuzzy expansion of Definition 4.3 by enabling it with the command recursive e-clauses.

Technically, adding as many rules as those defining $p$ for a predicate $q$ close to $p$ with a truth degree $\alpha$ might develop a space issue for a high number of rules. Fortunately, thanks to the tabling-based operational mechanism of FuzzyDES, it is possible to simplify this transformation. So, $q$ can be defined as a straight call to $p$ with a truth degree $\alpha$ (see the second item of the next definition).

**Definition 4.3:** Let $\Pi$ be a logic program, $\mathcal{R}$ a similarity relation on the syntactic domain generated by $\Pi$, and $\lambda \in [0, 1]$ a cut value. For each predicate $p$ defined in $\Pi$,  
1) for each clausal rule $(p(t_1, \ldots, t_n) \leftarrow Q, \delta)$ in $\Pi$ defining the $n$-ary predicate $p$, add to the transformed program $\Pi'$ the e-clause:
$$p(x_1, \ldots, x_n) \leftarrow \delta \land x_1 \approx t_1 \land \cdots \land x_n \approx t_n \land Q.$$  
2) for each entry $\mathcal{R}(p, q) = \alpha \geq \lambda$ in $\mathcal{R}$ (with $q \neq p$) add to the transformed program $\Pi'$ the e-clause:
$$q(x_1, \ldots, x_n) \leftarrow \alpha \land p(x_1, \ldots, x_n).$$

where each $x_i$ is a fresh variable.

Since this transformation is specific of the FuzzyDES system, we name it (for short) FDES expansion, and the transformed programs generated by its application FDES expanded programs.

Under this definition, if a program with $n$ predicates, each one defined by $r_i$ rules, and being it close to other $c_i$ predicates, then last program transformation generates $\sum_{i=1}^{n} r_i + c_i$ rules. However, the program transformation of Definition 4.2 generates $\sum_{i=1}^{n} r_i$ rules. Therefore, many transformed rules can be saved for a proximity (or similarity) relation with a huge amount of defined rules.

Note that the transformation of Definition 4.3 is only viable in the context of languages, such as Datalog, with specific syntactic constraints and using a tabling-based computation strategy. However, languages that follow a top-down strategy, like Bousi-Prolog, cannot benefit from this transformation. Note that this transformation can convert a program with a finite search space into one with an infinite search space for a pure top-down computational strategy (for instance, think of $\mathcal{R}(p, q) = \alpha_1$ and $\mathcal{R}(q, p) = \alpha_2$ that would lead to mutually recursive e-clauses).

The following example shows a transformation following Definition 4.3 by enabling it with the command fuzzy_expansion des.

**Example 2:** Consider again the program fragment of Example 1. After the execution of the predicate expand_rule/7 the following Datalog code is generated:
$$\text{r(A, D)} :- \text{'$unify_arguments'([[[A,a,B]]]),}$$
$$\text{p(A, D)} :- \text{'$t_norm'('\cdot,[[D,B]],D).}$$
$$\text{p(A, D)} :- \text{'$unify_arguments'([[A,X,B]],r(X,C)),}$$
$$\text{p(A, D)} :- \text{'$t_norm'('\cdot,[[D,0.8,C,B]],D).}$$
$$\text{q(A, D)} :- \text{'$unify_arguments'([[A,c,B]]),}$$
$$\text{q(A, D)} :- \text{'$over_lambda_cut'(0.6), q(A,B),}$$
$$\text{q(A, D)} :- \text{'$over_lambda_cut'(0.6), p(A,B),}$$
$$\text{q(A, D)} :- \text{'$t_norm'('\cdot,[[0.6,B]],D).}$$

Solving e-clauses from this transformation in the deductive setting poses no non-termination problems.

Finally, as we shall justify in Section V, the Datalog code generated by the transformation of either Definition 4.2 or 4.3 (depending on the flag fuzzy_expansion) is executed by the deductive engine of the FuzzyDES system, therefore emulating the result of executing the original program under WSLD resolution.

**F. Fuzzy Answer Subsumption**

As introduced in Subsection III-D, answer subsumption in a deductive database simply resorts to a membership test. Indeed, this approach can be also used in the fuzzy setting, but a different, more-efficient notion of answer subsumption can be applied as we propose at the end of this subsection.

Following any of the expansions explained in the former Subsection IV-E, each program rule is compiled to a rule with an additional argument for the approximation degree. FuzzyDES handles a user query $Q$ by transforming the query into an autoview for the predicate answer $m+1$, where its body is the compiled query, and $m$ are the relevant variables in $Q$. The last argument of the transformed call is the placeholder for the approximation degree. The outcome to the user query is thus built by solving the call answer($X_1, \ldots, X_m, X_0$), where $X_1, \ldots, X_m \subseteq \text{vars}(Q)$ are the relevant variables in the user query, and $X_0$ is the variable for the approximation degree. Each matching tuple $t_1$ in the answer table (i.e., $t_1=\text{answer}(X_1, \ldots, X_m, X_0, \sigma_1)$) is presented to the user as $\text{answer}(X_1, \ldots, X_m, \sigma_1)$ with $X_0 \sigma_1$. The answer table holds entries for all user predicates involved in solving a user query, each one with the last argument being the approximation degree, which is ground as the other arguments are. Then, along solving, it is possible to deduce a tuple $t_1 \equiv t(\overline{C}, D)$ so that there already exists another tuple $t_2 \equiv t(\overline{C}, D')$ in the answer table, differing only in the approximation degree. If $D' > D$, it is not worthwhile to add the same tuple with a lower approximation degree, so that we say that $t_2$ subsumes $t_1$. This can be seen as a generalization of answer subsumption to the fuzzy setting, and we apply in the deductive setting obtaining at least three advantages: First, the size of the answer table is reduced. Second, answer table look-ups are therefore more efficient. And, third, joins are simplified by taking into account less tuples.

**V. OPERATIONAL SEMANTICS FOR EXPANDED PROGRAMS**

This section formally describes, at a high abstraction level, the operational semantics for expanded programs implemented both by Bousi-Prolog and the FuzzyDES systems. This semantics simulates the WSLD resolution rule of Definition 4.1,
and is the basis to the implementation described in Subsection IV-E. The main goal of this section is to establish that, in fact, the behavior of a program executed by using WSLD resolution is equivalent to one of the corresponding expanded program when it is executed by this abstract operational mechanisms.

In the remainder of this section we shall work inside the framework of the extended language built by e-clauses and e-goals. $\Pi'$ denotes a transformed program (or expanded program) obtained by applying either Definition 4.2 or Definition 4.3 on a logic program $\Pi$ equipped with a similarity relation $R$ and a cut value $\lambda$. What follows, transition steps are applied to underlined fragments, and the symbols $Q$, $\delta$ are denoting conjunction of atoms and, possibly, approximation degrees.

Definition 5.1: We define the operational semantics for expanded programs as a transition system $(E, \Rightarrow_{Ex})$ where $E$ is a set of triples $\{(\langle \langle \wedge \delta \rangle \rangle, \theta, \alpha) \}$ (e-goal, substitution, approximation degree), and the transition relation $\Rightarrow_{Ex} \subseteq (E \times E)$ is the smallest relation which satisfies:

**Rule 1:** if $\beta \in (0, 1]$, and $(\delta, \sigma) \geq \lambda$, $\langle \langle \beta \wedge (Q') \rangle, \theta, \alpha \rangle \Rightarrow_{Ex} \langle \langle Q', \beta \delta \sigma \rangle \rangle$

**Rule 2:** if $\sigma = \text{wmg}_{\Pi}(A, B) \neq \text{fail}$, $\beta = R(A, \sigma, B, \sigma) \geq \lambda$, and $(\delta, \sigma) \geq \lambda$, $\langle \langle A \approx B \wedge (Q') \rangle, \theta, \alpha \rangle \Rightarrow_{Ex} \langle \langle Q', \sigma, \beta \delta \sigma \rangle \rangle$

**Rule 3:** if $(p(x_1, \ldots, x_n) \leftarrow \beta \wedge x_1 \approx t_1 \wedge \cdots \wedge x_n \approx t_n \wedge (Q')) \ll \Pi'$, $\langle \langle \text{t}, \text{t} \rangle, \theta, \alpha \rangle \Rightarrow_{Ex} \langle \langle \beta \wedge \sigma_1 \approx t_1 \wedge \cdots \wedge \sigma_n \approx t_n \wedge (Q''), \theta, \alpha \rangle \rangle$

Note that in the operational step defined by Rule 3, we perform a syntactic unification of the selected atom of the e-goal and the head of the e-clause. Note also that, in this case, the most general unifier $\{x_1/s_1, \ldots, x_n/s_n\}$ does not participate in the final computed answer because its domain variables are standardized apart (i.e., they are fresh variables) and does not affect the bindings of the substitution $\theta$. In what follows, we will often speak of simplification steps to refer to the steps performed with Rule 1 above.

To achieve a more condensed notation in the proofs, throughout the remainder of this section, we introduce the following notation: we write $\sigma_\Pi$ for the sequence of syntactic objects $o_1, \ldots, o_n$; similarly, $\sigma_\Pi$ denotes the composition of substitutions $\sigma_1 \sigma_2 \cdots \sigma_n$.

A. Semantic Equivalence for BPL Expanded Programs

In the sequel, we prove the semantic equivalence between the WSLD rule, applied to a logic program $\Pi$, and the operational mechanism of Definition 5.1, when it is applied to the BPL expanded program $\Pi'$ (obtained from $\Pi$ and generated by Definition 4.2).

First we recall a useful property of a similarity relation $R$ proved by M. Sessa.

**Proposition 5.2:** [7, pag. 397] Let $R$ be a similarity relation and $\lambda > 0$ a cut value. For any substitution $\theta$ and terms $t, t'$, if $R(t, t') \geq \lambda$ then $R(t\theta, t'\theta) = R(t, t') \geq \lambda$.

**Lemma 5.3:** Given a program $\Pi$ with a similarity relation $\mathcal{R}$, an associated t-norm $\Delta$, and a cut value $\lambda \in (0, 1]$, let $\Pi'$ be the BPL expanded program. If there is a step $S$:

$$\langle \langle (\langle \pi \rangle \wedge (Q'), \theta, \alpha \rangle \rangle, \theta, \alpha \rangle \Rightarrow_{\text{wmg}_{\Pi}} \langle \langle (\pi \wedge (Q') \sigma), \theta, \gamma \delta \rangle \rangle$$

in $\Pi'$, then there is a derivation:

$$\langle \langle (\pi \wedge (Q') \sigma), \theta, \gamma \delta \rangle \rangle \Rightarrow_{\text{Ex}} \langle \langle (Q \wedge (Q') \sigma), \theta, \gamma \delta \rangle \rangle$$

in $\Pi'$, which computes the same state. PROOF. If there is a step $S$ in $\Pi'$, is because there exists a graded rule $C = \langle (p(t n) \langle \Pi \rangle) \Rightarrow \theta \rangle$ in $\Pi$ such that

$$\sigma = \text{wmg}_{\Pi}(q(p(t n)), (\Pi \sigma)) = \text{wmg}_{\Pi}(s n = t n),$$

with approximation degree

$$\lambda = R(q(p(t n)) \sigma, (\Pi \sigma)) = R(q, p) \sigma, (\Pi \sigma) \geq \lambda$$

and $\gamma = \Delta \lambda \geq \lambda$.

On the other hand, by Definition 4.2, if $R(p, q) = \beta \geq \lambda$, there is an e-clause $\langle (p(t n)) \leftarrow (\delta \lambda) \wedge \pi n \approx t n \wedge (Q') \rangle \ll \Pi'$. Therefore, it is possible to construct the following derivation in $\Pi'$:

$$\langle \langle (\pi) \wedge (Q') \rangle, \theta, \alpha \rangle \Rightarrow_{\text{Ex}} \langle \langle (\pi \wedge (Q') \sigma), \theta, \gamma \delta \rangle \rangle$$

where $\sigma_1 = \text{wmg}_{\Pi}(\pi_1 \approx t_1)$, for $2 \leq i \leq n$, $\sigma_i = \text{wmg}_{\Pi}(\pi_i \approx t_i \pi_{i-1})$, and, thanks to Proposition 5.2, $R(s_i \pi_i, \pi_{i-1}) = R(t_i, \sigma_i, \sigma_{i-1}) \approx \pi_i$.

Note that, when we apply the weak unification algorithm described in Section IV-A to the unification problem $\{\pi n \approx t n\}$, we reach a successful configuration:

$$\langle \langle s_1 \approx t_1, \ldots, s_n \approx t_n \rangle, \ldots \rangle \Rightarrow_{\text{Ex}} \langle \langle (Q \wedge (Q') \sigma), \theta, \gamma \delta \rangle \rangle$$

and this weak unification process closely follows the steps performed with the subgoals $s_i \approx t_i$ in the former derivation with the BPL expanded program. Therefore, $\sigma = \text{wmg}_{\Pi}(\pi n \approx s n) = \sigma$ and $R(s n, \pi n \approx s n) \approx \Delta_{i=1}^{n} \pi_i$.

The following proposition establishes a kind of completeness result where we prove that derivations in the original program using the WSLD resolution rule can be reproduced by the FuzzyDES operational mechanism in the transformed program.

**Proposition 5.4:** Given a program $\Pi$ with a similarity relation $R$, let $\Pi'$ be the BPL expanded program. If there
is a derivation \( D = (\langle \neg Q, \theta, \alpha \rangle \Rightarrow \text{WSLD}^* \langle \neg Q', \theta', \alpha' \rangle) \) in \( \Pi \), then there is a derivation \( \langle \neg Q, \theta, \alpha \rangle \Rightarrow \text{Ex}^* \langle \neg Q', \theta', \alpha' \rangle \) in \( \Pi' \), which computes the same state.

**Proof.** By induction on the length of the derivation \( D \) and Lemma 5.3.

Now we proceed by demonstrating the reverse of the last proposition, which constitute a kind of soundness result.

**Lemma 5.5:** Given a program \( \Pi \) with a similarity relation \( \mathcal{R} \), an associated t-norm \( \triangle \), and a cut value \( \lambda \in (0, 1] \), let \( \Pi' \) be the BPL expanded program. If there is a derivation \( \Pi' : \langle \neg p(t_n) \wedge Q', \theta, \alpha \rangle \Rightarrow \text{Ex}^* \langle \neg (Q \wedge Q') \sigma, \theta, \lambda \Delta \alpha \rangle \) in \( \Pi' \), then there is a step \( S' : \langle \neg p(t_n) \wedge Q', \theta, \alpha \rangle \Rightarrow \text{WSLD}^* \langle \neg (Q \wedge Q') \sigma, \theta, \lambda \Delta \alpha \rangle \) in \( \Pi \).

**Proof.** Note that we can assume that the shape of the derivation \( D' \) is:

\[
\langle \neg (p(t_n) \wedge Q') \wedge \theta, \alpha \rangle
\Rightarrow_{\text{Ex}}^* \langle \neg (\beta \wedge (Q \wedge Q') \sigma), \theta, \Delta \alpha \rangle
\Rightarrow_{\text{Ex}} \langle \neg (Q \wedge Q') \sigma, \theta, \lambda \Delta \alpha \rangle
\]

where the first step is performed by Rule 3 and then it is followed by a sequence of applications of Rule 2, with \( \lambda_i = \mathcal{R}(t_i, s_i, \sigma) \) and \( \sigma = \text{wmgu}_k^L((t_n \approx s_n)) \). Finally, a simplification step with Rule 1 is performed and the degree \( \beta \) is compounded with \( \Delta \lambda_{n+1} \lambda_i \) to obtain \( \lambda' \).

If the first step of derivation \( D' \) is possible, it is because there exists an e-clause \( C' = (p(t_n) \leftarrow (\delta \wedge Q') \wedge \mathcal{R}(t_n \approx t_q \wedge Q)) \) in \( \Pi' \) and there must be an entry \( \mathcal{R}(q, p) = \beta \geq \lambda \) in \( \mathcal{R} \). So, there exists a clause \( C = (q(t_n) \leftarrow Q, \delta) \) in \( \Pi \), whose head weakly unify with \( p(t_n) \) and \( \text{wmgu}_k^L(q(t_n), p(t_n)) = \text{wmgu}_k^L(t_n \approx s_n) = \sigma \) with approximation degree \( \mathcal{R}(q(t_n) \sigma, p(t_n) \sigma) = \mathcal{R}(p, q) \Delta (\Delta \lambda_{n+1} \mathcal{R}(t_n, s_n) \sigma) = \beta \Delta (\Delta \lambda_{n+1} \lambda_i) = \lambda' \). Therefore, it is possible the WSLD step \( S' \).

**Proposition 5.6:** Given a program \( \Pi \) with a similarity relation \( \mathcal{R} \), let \( \Pi' \) be the BPL expanded program. If there exists a derivation \( \Pi' : (\langle \neg Q, \theta, \alpha \rangle \Rightarrow \text{WSLD}^* \langle \neg Q', \theta', \alpha' \rangle) \) in \( \Pi' \), then there exists a derivation \( \langle \neg Q, \theta, \alpha \rangle \Rightarrow \text{WSLD}^* \langle \neg Q', \theta', \alpha' \rangle \) in \( \Pi \), which computes the same state.

**Proof.** By induction on the length of the derivation \( \Pi' \). Without loss of generality we can assume that the steps in derivation \( \Pi' \) are conveniently ordered to allow the application of Lemma 5.5.

The last proposition jointly with Proposition 5.4 state the equivalence of both operational mechanisms and the correctness of our implementation.

**B. Semantic Equivalence for FDES Expanded Programs**

In this subsection we turn our attention to FDES expanded programs. The ultimate objective, as in the previous subsection, is to prove the equivalence of WSLD resolution and the source program \( \Pi \) with respect to the operational mechanism of Definition 5.1, when it is applied to an FDES expanded program \( \Pi' \) obtained from \( \Pi \). In this case, we proceed in an indirect way, studying the relation between these two types of transformed programs, and proving that they are semantically equivalent (modulo subsumed answers) with respect to the operational semantics for expanded programs of Definition 5.1.

Given a source program \( \Pi \), we first prove that the BPL expanded program \( \Pi' \) for \( \Pi \) results from the unfolding transformation of the corresponding FDES expanded program \( \Pi'' \).

**Program transformation** is an optimization technique for computer programs that, from an initial program \( \Pi_0 \), derives a sequence \( \Pi_1, \ldots, \Pi_n \) of transformed programs by applying elementary transformation rules which improve the original program under some criteria.

Unfolding is a well-known semantics-preserving program transformation rule (first introduced in [27] to optimize functional programs). In essence, it is usually based on the application of operational steps on the body of program rules. The unfolding transformation is able to improve programs, generating more efficient code. Unfolding is the basis for developing sophisticated and powerful programming tools, such as unfolding/unfolding transformation systems and partial evaluators.

In our fuzzy framework, the unfolding transformation can be defined, similarly to [28], as follows:

**Definition 5.7:** Let \( \Pi' \) be an expanded program and \( \mathcal{R} : (A \leftarrow Q) \in \Pi' \) a (non unit) program rule. Then, the fuzzy unfolding of \( \Pi' \) with respect to the rule \( \mathcal{R} \) and a fixed selection rule is the new expanded program \( \Pi'' = (\Pi' \cup \mathcal{U}) \setminus \{\mathcal{R}\} \) such that:

\[ \mathcal{U} = \{A \sigma \leftarrow \alpha \wedge Q' \mid \{Q'; id; 1\} \Rightarrow \text{Ex} \langle Q'; \sigma; \alpha \rangle \} \]

In order to accelerate the unfolding transformation process, we will allow to perform a sequence of simplification steps after an unfolding step.

A BPL expanded program can be obtained by unfolding the FDES expanded program. Next proposition states this result.

**Proposition 5.8:** Given a program \( \Pi \) with a similarity relation \( \mathcal{R} \), an associated t-norm \( \Lambda \), and a cut value \( \lambda \in (0, 1] \), let \( \Pi' \) be the BPL expanded program. The BPL expanded program \( \Pi'' \) can be obtained by unfolding \( \Pi' \), after disregarding rules that produce subsumed answers.

**Proof.** Without loss of generality, we can assume a program \( \Pi \) defining two predicates \( p \) and \( q \), each one defined by \( k \) and \( l \) rules respectively, and equipped with a similarity relation \( \mathcal{R} \) (characterized by the reflexive, symmetric, transitive t-closure of the entries \( \mathcal{R}(p, q) = \alpha, \mathcal{R}(q, r) = \beta \) and \( \mathcal{R}(p, r) = \gamma \)). Note that since the predicate \( r \) is not defined by rules, there are no \( e \)-clauses for it.

\[ \Pi = \{R_{p_1} : \langle p(t_{1n}) \Leftarrow \mathcal{Q}_1, \delta_{p_1} \rangle, \]
\[ R_{p_k} : \langle p(t_{kn}) \Leftarrow \mathcal{Q}_k, \delta_{p_k} \rangle, \]
\[ R_{q_1} : \langle q(t_{1n}) \Leftarrow \mathcal{B}_1, \delta_{q_1} \rangle, \]
\[ R_{q_l} : \langle q(t_{ln}) \Leftarrow \mathcal{B}_l, \delta_{q_l} \rangle \} \]

For this program, the FDES expanded program is:

\[ \Pi_1 = \{R_{p_1}^1 \leftarrow p(t_{1n}) \Leftarrow \delta_{p_1} \wedge \mathcal{Q}_1, \]
\[ R_{p_k}^1 \leftarrow p(t_{kn}) \Leftarrow \delta_{p_k} \wedge \mathcal{Q}_k, \]
\[ R_{p,q} \leftarrow q(t_{1n}) \Leftarrow \alpha \wedge p(t_{1n}), \]
\[ R_{p,r} \leftarrow r(t_{1n}) \Leftarrow \gamma \wedge p(t_{1n}), \]
\[ R_{q_1}^1 : q(\overline{x}_n) \leftarrow \delta_{q_1} \land \overline{x}_n \approx t_{1n} \land B_1, \]
\[ \ldots \]
\[ R_{q_k}^1 : q(\overline{x}_n) \leftarrow \delta_{q_k} \land \overline{x}_n \approx t_{kn} \land B_1, \]
\[ R_{q,p}^\prime : p(\overline{x}_n) \leftarrow \alpha \land q(\overline{x}_n), \]
\[ R_{q,r}^\prime : r(\overline{x}_n) \leftarrow \beta \land q(\overline{x}_n) \]

Now, we proceed to unfold \( \Pi^1 \) with respect to \( R_{p,q} \) using a selection rule that selects the subgoal \( p(\overline{x}_n) \), in the body of \( R_{p,q} \). Then, for each rule \( R_{p,i}^1 \) in \( \Pi^1 \):

\[ \langle \alpha \land p(\overline{x}_n), id, 1 \rangle \xrightarrow{R_{p,i}^1} (\alpha \land \delta_{p_i} \land \overline{x}_n \approx t_{1n} \land Q_i, id, 1) \]
\[ \xrightarrow{\text{Simp}^+} (\overline{x}_n \approx t_{1n} \land Q_i, id, \alpha \delta_{p_i}) \]

leading, by Definition 5.7, to the rules:

\[ q(\overline{x}_n) \leftarrow (\alpha \delta_{p_i}) \land \overline{x}_n \approx t_{1n} \land Q_i \]

For the rule \( R_{q,p} \), also defining \( p \), we have the derivation

\[ \langle \alpha \land p(\overline{x}_n), id, 1 \rangle \xrightarrow{R_{q,p}^\prime} (\alpha \land \alpha \land q(\overline{x}_n), id, 1) \]
\[ \xrightarrow{\text{Simp}^+} (q(\overline{x}_n), id, \alpha) \]

However, this last derivation leads to the recursive rule \( q(\overline{x}_n) \leftarrow \alpha \land q(\overline{x}_n) \) which can be removed since it does not provide new answers. Therefore,

\[ U^1 = \{ R_{q_1}^2 : q(\overline{x}_n) \leftarrow (\alpha \delta_{p_1}) \land \overline{x}_n \approx t_{1n} \land Q_1, \]
\[ \ldots \]
\[ R_{q_k}^2 : q(\overline{x}_n) \leftarrow (\alpha \delta_{p_k}) \land \overline{x}_n \approx t_{kn} \land Q_k \} \]

with this first unfolding step we obtain an unfolded program \( \Pi^2 = (\Pi^1 \cup U^1) \setminus \{ R_{p,q} \} \). Note that the rule \( R_{p,q} \) in \( \Pi^2 \) has been removed and the rules of the BPL expanded program produced by the \( k \) rules defining \( p \) in \( \Pi \) and the entry \( \mathcal{R}(p, q) = \gamma \) (see Definition 4.2) have been added.

Starting from \( \Pi^2 \), we can undertake a new unfolding step. In this case, we proceed to unfold \( \Pi^2 \) with respect to \( R_{p,r} \). Again, selecting the subgoal \( p(\overline{x}_n) \), in the body of \( R_{p,r} \). Then, for each rule \( R_{p,i}^2 \) in \( \Pi^2 \):

\[ \langle \gamma \land p(\overline{x}_n), id, 1 \rangle \xrightarrow{R_{p,i}^2} (\gamma \land \delta_{p_i} \land \overline{x}_n \approx t_{1n} \land Q_i, id, 1) \]
\[ \xrightarrow{\text{Simp}^+} (\overline{x}_n \approx t_{1n} \land Q_i, id, \gamma \delta_{p_i}) \]

leading, by Definition 5.7, to the rules:

\[ r(\overline{x}_n) \leftarrow (\gamma \delta_{p_i}) \land \overline{x}_n \approx t_{1n} \land Q_i \]

For the rule \( R_{q,p} \), also defining \( p \), we have the derivation

\[ \langle \gamma \land p(\overline{x}_n), id, 1 \rangle \xrightarrow{R_{q,p}^\prime} (\gamma \land \gamma \land q(\overline{x}_n), id, 1) \]
\[ \xrightarrow{\text{Simp}^+} (q(\overline{x}_n), id, \gamma \alpha) \]

However, this last derivation leads to the recursive rule \( r(\overline{x}_n) \leftarrow (\gamma \alpha) \land q(\overline{x}_n) \). It is important to note that \( \beta \geq (\gamma \alpha) \), thanks to the transitive property of the similarity \( \mathcal{R} \).

Hence, we can remove the preceding unfolded rule since it does not provide better answers than the rule \( R_{q,r} \) which is subsuming it. Therefore,

\[ U^2 = \{ R_{r_1}^2 : r(\overline{x}_n) \leftarrow (\gamma \delta_{p_1}) \land \overline{x}_n \approx t_{1n} \land Q_1, \]
\[ \ldots \]
\[ R_{r_k}^2 : r(\overline{x}_n) \leftarrow (\gamma \delta_{p_k}) \land \overline{x}_n \approx t_{kn} \land Q_k \} \]

and we obtain an unfolded program \( \Pi^3 = (\Pi^2 \cup U^2) \setminus \{ R_{p,r} \} \). Once again, the rule \( R_{p,r} \) has been removed and the rules of the BPL expanded program produced by the \( k \) rules defining \( p \) in \( \Pi \) and the entry \( \mathcal{R}(p, r) = \gamma \) (see Definition 4.2) have been added.

In a complete similar way we can unfold \( \Pi^3 \) w.r.t to \( R_{q,r} \) given the unfolded program \( \Pi^4 \) and in a further unfolding transformation we can unfold \( \Pi^4 \) w.r.t to \( R_{q,r} \) given the unfolded program \( \Pi^5 \) where the rules \( R_{q,p} \) and \( R_{q,r} \) have been removed and the rules of the BPL expanded program produced by the \( l \) rules defining \( q \) in \( \Pi \) and the entries \( \mathcal{R}(q, p) = \alpha \) and \( \mathcal{R}(q, r) = \beta \) (see Definition 4.2) have been added. As a result \( \Pi^5 \) is the BPL expanded program.

In summary, starting from the FDES expanded program, \( \Pi^1 \), we carry out continuous unfolding transformation steps, leading to a sequence of unfolded programs: \( \Pi^2, \Pi^3, \Pi^4, \Pi^5 \), which ends into the BPL expanded program for \( \Pi \).  

Finally, we can prove that an FDES expanded program executed by the operational mechanism of Definition 5.1 is semantically equivalent to the original program executed by WSLD resolution.

**Proposition 5.9:** Given a program \( \Pi \) with a similarity relation \( \mathcal{R} \) and a cut value \( \lambda \in (0, 1) \), let \( \Pi' \) be the FDES expanded program. \( \Pi' \), executed by the operational mechanism of Definition 5.1, is semantically equivalent to \( \Pi \) executed by WSLD resolution. That is, they produce the same fuzzy computed answers (modulo subsumed answers).

**Proof:** Given a program \( \Pi \) and its corresponding FDES expanded program \( \Pi' \), by Proposition 5.8, the BPL expanded program \( \Pi'' \) is the unfolded program obtained from \( \Pi' \) if some rules that produce subsumed answers are removed. First note that, unfolding is a semantics-preserving program transformation (preserving fuzzy computed answers). On the other hand, according to propositions 5.6 and 5.4, the BPL expanded program \( \Pi'' \), executed by the operational mechanism of Definition 5.1, is semantically equivalent to the original program executed by WSLD resolution. Therefore, we can affirm that an FDES expanded program executed by the operational mechanism of Definition 5.1 is semantically equivalent to the original program executed by WSLD resolution (modulo subsumed answers).

**VI. Performance**

This section tries to highlight the effect of different parameters of the system with respect to their impact on its performance. Solving time and data structure sizes are elements of interest in performing experiments. A number of tests have been executed with different parameters to analyse scalability. As test platform, we used a Windows 10 64 bit OS on an Intel Xeon CPU E3-1505M v5 (4 physical cores) running at 2.8 GHz at its peak, with 16GB RAM. DES (programmed in Prolog) is run with SICStus Prolog 4.3.1 64 bit in interpreted mode (no compiled executable). Solving time is only due to solving, eliding e.g. parsing and display. Statistics in DES are enabled to collect data as timings, look-ups, and data structure sizes. This incurs in a small but noticeable overhead on the
system performance. For the fuzzy environment, we consider a similarity (transitive) relation \(\sim\) with a product t-norm.

Regarding solving parameters, we analyse the impact of the two different expansions BPL and FDES. The next two subsections focus on this: the first one on extensional predicate similarity, and the second one on intensional predicate similarity. We do not handle the case of constant similarity because it does not affect expansions, so that the same performance results would be retrieved for both. Subsection VI-C analyses the performance impact of enabling answer subsumption for both expansions. Finally, Subsection VI-D analyses the performance impact of specifying a \(\lambda\)-cut for both expansions.

### A. Comparing Expansions for Extensional Predicate Similarity

A first experiment consists of examining the performance of the two expansions BPL and FDES for extensional predicates (consisting only of facts). Here, we consider \(n\) extensional predicates so that they are similar as follows: \(p_i/2 \sim p_{i+1}/2 = 0.5\) for \(i \in \{1, \ldots, n-1\}\). Each predicate is defined by \(m\) facts of the form: \(p_i(c_{i+j-1}, c_{i+j-1})\), where \(i \in \{1, \ldots, n\}\) and \(j \in \{1, \ldots, m\}\). For example, for \(n = 2\) and \(m = 3\), the resulting test program is:

\[
\begin{align*}
& p_1(1,2), p_1(2,3), p_1(3,4), \\
& p_2(2,3), p_2(3,3), p_2(4,6).
\end{align*}
\]

This way, a pair of constants which is not in a predicate \(p_i\), but is in \(p_{i+1}\) can be deduced for \(p_i\) with an approximation degree less than one, and as \(p_0\) and \(p_1\) parameter instances we selected \(n\) predicate instances so that they are similar as follows: \(p_i/2 \sim p_{i+1}/2 = 0.5\) for \(i \in \{1, \ldots, n-1\}\). Each predicate is defined by \(m\) facts of the form: \(p_i(c_{i+j-1}, c_{i+j-1})\), where \(i \in \{1, \ldots, n\}\) and \(j \in \{1, \ldots, m\}\). For example, for \(n = 2\) and \(m = 3\), the resulting test program is:

\[
\begin{align*}
& p_1(1,2), p_1(2,3), p_1(3,4), \\
& p_2(2,3), p_2(3,3), p_2(4,6).
\end{align*}
\]

For \(n = 2\), the solving speed-up fluctuates for small \(m\) up to 2,000. Note that solving times for these values of \(m\) are rather low (less than 32 ms) so that OS mostly interferes for these numbers. For larger numbers, up to a solving speed-up of 1.39x is got. Observe that only two extensional predicates are made similar. The ratio of rules increases until the asymptote 2.00 is reached (BPL generates twice the rules than FDES). Before this, for small number of \(m\), the threshold which corresponds to the fixed number of rules for computing the t-closure is noticeable when computing the ratio. For example, the number 1.62 is computed as \((40 + 7)/(22 + 7)\), where 40 is the number of compiled rules for the predicates \(p_1\) and \(p_2\) for BPL expansion, 22 is for FDES, and 7 are the rules needed to compute the t-closure (in this case, the t-closure delivers only 4 equations). Considering higher values for \(n\), an increasing solving speed-up greater than 1 for increasing \(m\)'s is found for \(m = 200\) and \(m = 5,000\) for \(n = 4\) and \(n = 6\), respectively. Recall that the FDES expansion resists to convert any predicate (whether recursive or not) to a recursive predicate. However, the implementation of DES has no optimization for recursion, as the differential semi-naive recursion optimization. This makes harder to solve recursive predicates than non-recursive ones, and as \(n\) increases, so the number of recursive predicates does. However, for given numbers of \(n\) and \(m\) there is a point in which this cost is overcome by the cost of handling a greater number of rules per predicate in the database. For example, for \(n = 4\) and \(m = 200\) there are 3,209 rules for BPL vs. 821 rules for FDES (which makes a ratio of 3.91), and for \(n = 6\) and \(m = 5,000\) there are 180,011 rules for BPL vs. 30,041 rules for FDES (ratio of 5.99). The cost of consulting rules notably increases mainly because of two factors: First, the database is dynamic and not indexed, so that when adding new rules, the database must be sequentially explored. Second, PDG construction is implemented with an exponential-time algorithm. Following the results in this experiment, two issues are amenable to be improved: recursion optimization, and consulting time (rule traversing and PDG construction).

### B. Comparing Expansions for Intensional Predicate Similarity

Following the comparison of both expansions, a second experiment considers a database composed of both extensional and intensional predicates (with non-empty rule bodies). We use the same parameters \(n\) and \(m\) as in last subsection, and introduce a new parameter \(k\). By contrast to the former experiment, here we focus on similarities between \(k\) intensional predicates out of \(n\). We denote the \(k\) intensional predicates as \(p_1, \ldots, p_k\), while the \(n - k\) extensional predicates are denoted as \(p_{k+1}, \ldots, p_n\). The form of each intensional predicate clause is \(p_i(X_1, X_{n-k+1}) := p_{k+1}(X_1, X_2), \ldots, p_n(X_{n-k}, X_{n-k+1})\), where \(i \in \{1, \ldots, k\}\).

Hence, intensional predicate bodies are defined as calls to each extensional predicate, correlated with shared variables so that a call \(p_{k+1}(X_1, X_{n+1})\) is followed by \(p_{k+1+1}(X_1, X_{n+2})\) for \(i < n - k\). As for the intensional predicate head, the first (resp. second) head variable is the first (resp. last) variable in the body. As it was just commented, similarity is only specified

### Table I

<table>
<thead>
<tr>
<th>(m)</th>
<th>(n = 2)</th>
<th>(n = 4)</th>
<th>(n = 6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.07  1.03  1.62</td>
<td>0.79  1.32  2.77</td>
<td>0.12  0.43  3.67</td>
</tr>
<tr>
<td>100</td>
<td>1.36  1.20  1.95</td>
<td>0.88  1.30  3.82</td>
<td>0.18  2.46  5.63</td>
</tr>
<tr>
<td>200</td>
<td>0.86  1.15  1.97</td>
<td>1.11  2.21  3.91</td>
<td>0.23  3.99  5.81</td>
</tr>
<tr>
<td>500</td>
<td>1.13  1.31  1.99</td>
<td>1.30  3.61  3.96</td>
<td>0.35  9.16  5.92</td>
</tr>
<tr>
<td>1,000</td>
<td>1.00  1.54  1.99</td>
<td>1.56  6.21  3.98</td>
<td>0.51  15.16 5.96</td>
</tr>
<tr>
<td>2,000</td>
<td>1.14  1.88  2.00</td>
<td>1.66  9.35  3.99</td>
<td>0.73  20.89 5.98</td>
</tr>
<tr>
<td>5,000</td>
<td>1.31  2.74  2.00</td>
<td>1.65  11.50 4.00</td>
<td>1.20  24.33 5.99</td>
</tr>
<tr>
<td>10,000</td>
<td>1.32  3.25  2.00</td>
<td>1.79  12.34 4.00</td>
<td>1.38  28.94 6.00</td>
</tr>
<tr>
<td>15,000</td>
<td>1.38  3.18  2.00</td>
<td>1.77  12.46 4.00</td>
<td>1.58  26.05 6.00</td>
</tr>
<tr>
<td>20,000</td>
<td>1.39  3.48  2.00</td>
<td>1.84  12.84 4.00</td>
<td>1.75  22.55 6.00</td>
</tr>
</tbody>
</table>
for intensional predicates. We assign, axiomatically, proximity equations for the $k$ intensional predicates. Specifically, we write $p_i/2 \cdot p_{i+1}/2 = 0.5$ for $i \in \{1, \ldots, k-1\}$. On the other hand, the rest of $n-k$ (extensional) predicates are defined as in the first experiment, i.e., each one defined with $m$ facts. For example, for $n=4$, $m=2$, $k=2$, the resulting test program is:

\[
p_{1/2} \cdot p_{2/2} = 0.5. \\
p1(X1,X3) :- p3(X1,X2), p4(X2,X3).
\]

We consider for the experiments 10 predicates ($n = 10$), $m \in \{10, 100, 200, 500, 1000, 2000, 5000, 10000, 15000, 20000\}$, and $k \in \{3, 5, 7\}$. The same goal $p1(a,b)$ has been considered.

Table II includes results for these parameters $m$ and $k$, and for each combination of them, two measures are shown: “So.” and “Co.” with the same meaning as before but with the ratios the other way round. Observe that the number of rules in the expansion coincides for both BPL and FDES (we therefore omit the column “Rs.”), and ranges from 87 to 140,017 (including the rules for t-closure computation).

As in the first experiment, fluctuations do occur for small solving times (less than 50 ms) related to small numbers for $m$. In this case, focusing on larger numbers of $m$, solving times are better in BPL than in FDES, and increase as $m$ grows. It is also observed that higher speed-ups are got for larger $k$ (number of intensional predicates). Though there are the same expanded rules in both cases, in FDES, there is one mutually recursive rule for each intensional predicate instead of a non-recursive rule in the case of BPL. Because (mutually) recursive rules are harder to solve in DES as already mentioned, this issue explains such behaviour. With respect to consult times, both expansions converge to 1 for larger numbers of $m$ because the same number of rules are generated. However, for small numbers of $m$, there is a timing threshold due to PDG construction, which follows a naïve algorithm. Again, recursive predicates make this algorithm to perform badly. This is also the reason for omitting PDG updating when submitting queries, which are automatic views (therefore rules added to the database and hence, the PDG should be updated). For the queries posed, there is no need for PDG update since in particular the stratification does not change.

### Table II

<table>
<thead>
<tr>
<th>$k = 3$</th>
<th>$k = 5$</th>
<th>$k = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>$m$</td>
<td>$m$</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.96</td>
</tr>
<tr>
<td>100</td>
<td>1.00</td>
<td>1.98</td>
</tr>
<tr>
<td>200</td>
<td>0.27</td>
<td>1.00</td>
</tr>
<tr>
<td>500</td>
<td>2.67</td>
<td>1.05</td>
</tr>
<tr>
<td>1000</td>
<td>1.94</td>
<td>1.01</td>
</tr>
<tr>
<td>2000</td>
<td>0.95</td>
<td>0.99</td>
</tr>
<tr>
<td>5000</td>
<td>1.28</td>
<td>1.03</td>
</tr>
<tr>
<td>10000</td>
<td>1.38</td>
<td>1.00</td>
</tr>
<tr>
<td>20000</td>
<td>1.35</td>
<td>0.97</td>
</tr>
</tbody>
</table>

#### C. Analysing Answer Subsumption

Subsection IV-F explained the concept of fuzzy answer subsumption to prune computations as an optimization to the naïve case. Here we test the benefits of this optimization with respect to both solving time and data size. By contrast to former experiments, the goal must succeed and return answer tuples to examine the amount of such tuples, the tuples in the extension table, and its solving time. Here we select the goal $p1(c1,X)$. Regarding the test program, we reuse the one in last subsection, which can be parametrized to specify a total number $n$ of predicates, the number of $k$ similar predicates, the number of $l$ similar constants, and the number of facts $m$ per extensional predicate.

Table III includes two test cases for $(n,m,k,l) \in \{(5,10,2,4),(7,100,4,10)\}$ and the expansions BPL and FDES (column Exp.). For the optimized case (answer subsumption enabled), it includes its solving speed-up (column S. S-U), the reduction of both the ET size (ET. Gain) and answer tuples (Ans. Gain).

Whereas in the BPL expansion, gains are not noticeable when enabling answer subsumption (because the BPL expansion avoids the generation of many redundant answers), in the FDES expansion great gains are achieved, both in solving time, ET size, and number of answers. Recall that the $t$-norm product improvement is achieved (even when answer subsumption was selected for experiments. Then, as the FDES expansion transforms non-recursive predicates related by similarity in mutually recursive predicates, recursive cycles are built along fixpoint iterations, producing the same tuples with decreasing similarity degrees. Thus, this optimization is a huge advantage in this scenario.

#### D. Analysing $\lambda$-cut

Finally, this subsection analyses the effect of the $\lambda$-cut value for both expansions. Again, we consider the same benchmark and parameters as in the last subsection with $(n,m,k,l) = (7,100,4,30)$, that is, we specify 4 similar intensional predicates (out of 7, with 3 extensional predicates each one with 100 facts) and 30 similar constants. This amount of similar constants will highlight the effect of the $\lambda$-cut for both expansions. The goal is the same as in the last subsection.

Table IV includes the column $\lambda$-cut for different selected values, showing results for both expansions BPL and FDES and both for solving time and ET size gains. These columns have the same meaning as in the last subsection.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>$k$</th>
<th>$l$</th>
<th>Exp.</th>
<th>S. S-U</th>
<th>ET. Gain</th>
<th>Ans. Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>BPL</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>4</td>
<td>10</td>
<td>DES</td>
<td>546</td>
<td>302</td>
<td>1,025</td>
</tr>
</tbody>
</table>
similarity are pruned. Note that in all cases, the number of answers is the same and, thus, the column “Ans. Gain” is elided. This improvement increases as λ-cut does. For the last value, there is no further computation cuts and the gains remain constant. It is also worth to note that the achievements are better in FDES than in BPL because more computations (due to the recursive part) are cut. Though, the computation time is still better in BPL.

VII. RELATED WORK

The idea of integrating concepts derived from fuzzy logic within a database is not new. In this section we give an abbreviated record of these achievements (see the supplementary material provided at IEEE Xplore for a more detailed study on fuzzy databases).

Buckles and Petry [29] fuzzify relational databases by associating similarity relations to the attribute domains in a database. In that way, they can weaken the equality relation in order to perform some kind of flexible querying through a extended relational algebra. Later on, Shenoi and Melton [30] extended this model to deal with proximity relations. Note that, although we share with these authors the use of similarity/proximity relations, the methods they use to represent and deal with the fuzzy component of a database differ by far from ours.

Umano [31] proposed an alternate model supported on the concept of possibility distribution. It was generalized by Prade and Testemale [32] and it turned into a mainstream in the area. FREDDI [10] synthesizes the most outstanding aspects of this prior approach but incorporating deductive capabilities. An extended implementation of the FREDDI architecture [11] adopts a clausal representation for the rules of the intensional database, and deductive capabilities are obtained by implementing a non-classical bottom-up algorithm inspired in Datalog techniques. Although this framework presents deductive characteristics, unlike ours, it requires the user, explicitly, to take into account the propagation of the truth degrees through an additional parameter that stores a “matching” degree, leading to a kind of hybrid system combining SQL and logic rules. In addition, though they claim that their proposal is based on two theoretical models, there is no theoretical provision for their combination. Moreover, this system does not seem to be available.

The introduction of fuzzy features in classical deductive databases, with Datalog as a query language, is a more recent activity and has not produced so many examples. In this field, some of the work has consisted in the addition of confidence factors or truth degrees to the facts and rules of the extensional and the intensional databases which are manipulated by fuzzy connectives (e.g., the work presented in [33]). However, we are interested in those works that introduce fuzzy characteristics within the framework of the Datalog language, primarily through the use of fuzzy relations. In this sense, the work of A. Achs et al. [9] can be considered as a pioneering work, since it integrated the use of fuzzy relations in the framework of the Datalog language for the first time. They started from an extended Datalog language, with truth degrees in facts and rules, to which they incorporated a similarity relation on the domain of attributes. Unfortunately, and contrary to our proposal, this work has the following limitations: i) it only considers the similarity between the attributes of a relation, not between the relation itself; and ii) Datalog programs undergo a transformation guided by the similarity relation, but this transformation is complex (and possibly inefficient) and does not adequately handle rules with non-variable arguments. Finally, to the best of our knowledge, there is no report of a publicly available implementation of their system.

The work of A. Achs et al. was developed in parallel and independently with respect to the works [34], [35] and [7]. There, they introduced similarity relations in a logic programming language, and the concept of unification by similarity was first developed. It is in this last stream, and especially in the works of M. Sessa, that we have found our inspiration. As within the Bousi-~Prolog language [5], [6], in the FuzzyDES system we make use of the unification algorithm proposed in [7]. However, there are some striking differences between the concepts presented in [7] and those we use: 7 i) We adopt a different notion of weak unifier and weak most general unifier. ii) Also, we have defined a more appropriate concept of similar clause, which has a beneficial impact on the efficiency of the system we have implemented. iii) We have introduced some new peculiarities into the operational semantics of our language (such as the use of a threshold or a different notion of family of computed answers). iv) Finally, we have developed new implementation techniques for efficient execution of programs. These conceptual and practical differences must be added to the differences arising from the fact that FuzzyDES uses an operational mechanism based on tabling (a bottom-up top-down-driven strategy) which have been adequately explained throughout this work.

VIII. CONCLUSIONS

This paper has introduced formal and practical aspects needed to transfer some techniques found in the fuzzy logic programming system Bousi-~Prolog into the deductive system DES. The new extension of DES, called FuzzyDES, defines a fuzzy Datalog language which is executed by the operational

---

### TABLE IV

<table>
<thead>
<tr>
<th>λ-cut</th>
<th>Exp.</th>
<th>S. S-U</th>
<th>ET. Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>BPL</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.05</td>
<td>BPL</td>
<td>93</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>120</td>
<td>11</td>
</tr>
<tr>
<td>0.10</td>
<td>BPL</td>
<td>146</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>190</td>
<td>16</td>
</tr>
<tr>
<td>0.25</td>
<td>BPL</td>
<td>247</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>316</td>
<td>27</td>
</tr>
<tr>
<td>0.50</td>
<td>BPL</td>
<td>790</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>492</td>
<td>60</td>
</tr>
<tr>
<td>0.75</td>
<td>BPL</td>
<td>494</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>738</td>
<td>443</td>
</tr>
<tr>
<td>0.90</td>
<td>BPL</td>
<td>494</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>738</td>
<td>443</td>
</tr>
</tbody>
</table>

7See [6] for a more extensive discussion of these differences.
mechanism of Weak SLD resolution. Our implementation performs this task by compiling FuzzyDES programs into core Datalog in order to obtain a set of Datalog rules that allows the emulation of the mentioned Weak SLD resolution procedure. We have precisely defined two processes for generating such compiled programs, leading to what we call the BPL expanded program and the FDES expanded program. By using program transformation techniques, we have studied the relationships between both expansions. Also we prove the semantic equivalence between the WSLD rule, applied to a logic program II, and the operational mechanism of Definition 5.1, when it is applied to a BPL or FDES expanded program II' obtained from II. Moreover, we performed an experimental analysis of the system for performance and scalability, comparing the proposed expansions. We have provided a working system able to solve queries and commands on fuzzy deductive databases and shown some of its features with an application example. Since the DES system has been used already for big data applications (deductive data warehouses and OLAP [36] and their performance [37]), we expect its fuzzy extension also amenable for such applications. To the best of our knowledge, this is the only publicly available system implementing an interactive fuzzy deductive database with user-defined fuzzy relations.

REFERENCES


