[Smu68] R.M. Smullyan. *First-order logic*. Berlin: Springer-Verlag, 1968.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, MD, 1988.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38:620 – 650, 1991.

References

[ABW88] K. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 89–142, Los Altos, CA, 1988. Morgan Kaufmann.

[AN78] H. Andreka and I. Nemeti. The generalized completeness of Horn predicate logic as a programming language. *Acta Cybernetica*, 4:3–10, 1978.

[Apt90] K. Apt. Logic programming. In J. van Leeuven, editor, *Handbook of theoretical computer science*, pages 493–574. MIT Press, Cambridge, MA, 1990.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41:181–229, 1990.

[AV91] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and data bases*, pages 293–322. Plenum Press, 1978.

[DG84] W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[EG92] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactuals. In *ACM Symposium on Principles of Database Systems*, pages 261–273, 1992.

[EHK81] R.L. Epstein, R. Haas, and R.L. Kramer. Hierarchies of sets and degrees below $0'$. In M. Lerman, J.H. Schmerl, and R.I. Soare, editors, *Logic Year 1979-80*, pages 32–48. Springer Verlag, 1981. S.L.N. in Mathematics 859.

[Fit85] M. C. Fitting. Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2:295–312, 1985.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th international symposium on logic programming*, pages 1070–1080, Cambridge, MA., 1988. MIT Press.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and P. Szeredi, editors, *Proceedings of the 7th international conference on logic programming*, pages 579–597, Cambridge, MA., 1990. MIT Press.

[MT91] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[MT93] W. Marek and M. Truszczyński. *Nonmonotonic logics; context-dependent reasoning.* Berlin: Springer-Verlag, 1993.

[MT94] W. Marek and M. Truszczyński. Revision specifications by means of revision programs. In *Logics in AI. Proceedings of JELIA '94.* Lecture Notes in Artificial Intelligence. Springer-Verlag, 1994.

[MW88] S. Manchanda and D.S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394, Los Altos, CA, 1988. Morgan Kaufmann.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

1. *For every stratification $\langle P_\xi \rangle_{\xi<\eta}$ of P and for every well-ordering $\prec$ of P which agrees with the stratification $\langle P_\xi \rangle_{\xi<\eta}$, the result of the sequential revision process for $\prec$ and D_I is the unique P-justified revision of D_I.*

2. *In, in addition, P and D_I are finite, the unique P-justified revision of D_I can be computed in time proportional to the total size of P and D_I.*

Lemma 5.2 together with Duality Theorem 2.4 allows for an estimate of arithmetic complexity of revisions by means of safe programs. The subsequent result is the analogue of the fundamental result on the complexity of the least model of the Horn logic program [Smu68, AN78].

We first need some definitions.

Definition 5.2 ([EHK81])

1. A subset $A \subseteq \omega$ is called a *d.r.e. set* (difference of r.e. sets) if there are r.e. sets B, C such that $A = B \setminus C$.

2. A subset $A \subseteq \omega$ is *weakly d.r.e.* if both A and $\omega \setminus A$ are d.r.e. sets.

The class of d.r.e. sets is not closed under complements in the very same way as r.e. sets are not closed under the complements. The weakly d.r.e sets play the role of "recursive" sets with respect to d.r.e. sets. More on d.r.e. sets in [EHK81].

We now have a basic result on the arithmetic complexity of revisions of recursive databases by means of recursive programs.

Theorem 5.3 *Let D be a recursive database and P be a recursive safe program. Then the result of P-justified revision of D is a weakly d.r.e. set.*

A converse result holds for a slightly modified class of revision programs.

Notice that the d.r.e. sets and their generalizations, $(n+1)$-r.e. sets form proper subclasses of the least field of sets containing the r.e. sets. This field, in turn, is properly contained in the field of Δ_2^0 sets of natural numbers. Thus revisions of recursive databases by means of safe programs determine rather restricted class of sets.

As noticed above the safe programs play the role of Horn programs in our theory. As in logic programming, some of useful properties of safe programs can be extended to a wider class of programs.

Definition 5.3 Let P be a revision program and let $\langle P_\xi \rangle_{\xi<\eta}$ be a partition of P. We say that $\langle P_\xi \rangle_{\xi<\eta}$ is a *stratification* of P if for every $\xi < \eta$:

1. P_ξ is safe, and

2. $head(P_\xi) \cap \bigcup_{\alpha<\xi} var(P_\alpha) = \emptyset$.

Safe programs are stratified. Notice also that revision programs obtained from locally stratified logic programs under the interpretation described in Section 2 are stratified according to Definition 5.3.

To test if a finite revision program P is stratified, one can use a modified version of the algorithm of Apt, Blair and Walker [ABW88]. It takes linear time in the size of P. Moreover, also in linear time, one can establish a partition of P into strata $\langle P_m \rangle_{m<n}$.

For stratified programs we have the following generalization of Theorem 5.1(1).

Theorem 5.4 *Let P be a stratified revision program. Then for every database D_I there exists a unique database D_R such that $\langle D_I, D_R \rangle$ is a P-justified transition.*

As in the case of safe programs, one can show that given a finite stratified revision program P and an initial database D, the unique P-justified revision of D can be computed in linear time (in the total size of P and D).

Consider a stratification $\langle P_\xi \rangle_{\xi<\eta}$ of a stratified program P. A well-ordering $\prec$ of P *agrees* with the the stratification $\langle P_\xi \rangle_{\xi<\eta}$ if whenever $\xi_1 < \xi_2 < \eta$ then every rule in P_{ξ_1} $\prec$-precedes every rule in P_{ξ_2} (such orderings exist). Now, we can generalize Theorem 5.1(3) and (4).

Theorem 5.5 *Let P be a stratified revision program and let D_I be a database. Then:*

(9) **if** $head(r) = \mathbf{in}(a)$ **then** $I := I \cup \{a\}$ **else if** $head(r) = \mathbf{out}(a)$ **then** $O := O \cup \{a\}$
(10) $R := R \cup \{r\}$
(11) $A := AR_P(D) \setminus R$
(12) **if** $I \cap O = \emptyset$ **and** $AR_P(D) = R$ **then** report "D is a P-justified revision of D_I"

While as stated, this algorithm is more complex than the previous one (the main loop has to be repeated $|P|!$ times), it can be improved. One can show that it is enough to consider only a subset of the set of all orderings of cardinality at most $2^{|P|}$.

Some minor improvements are also possible in algorithm **Guess_and_Check**. Despite all these improvements, both algorithms remain exponential. High complexity of computing justified revisions is a serious problem. Fortunately, there are wide classes of programs whose computational properties are much better. We discuss them in the next section.

5 Safe programs

We will now discuss two types of revision programs, *safe* and *stratified*, for which the task of finding a P-justified revision can be solved in polynomial time. Let us recall that we restrict our attention in the paper to propositional programs only. Results of this section can be partly extended to the predicate case. The details will be given in a full version of the paper.

Definition 5.1 A revision program P is *safe* if

1. there is no a such that $\mathbf{in}(a) \in var(P)$ and $\mathbf{out}(a) \in head(P)$
2. there is no a such that $\mathbf{out}(p) \in var(P)$ and $\mathbf{in}(a) \in head(P)$

For example, program $P_1 = \{\mathbf{in}(a) \leftarrow \mathbf{out}(b)\}$ is safe. Similarly, $P_2 = \{\mathbf{in}(a) \leftarrow \mathbf{out}(b), \mathbf{out}(c) \leftarrow, \mathbf{out}(d) \leftarrow \mathbf{in}(a), \mathbf{out}(b) \leftarrow\}$ is also safe. Program $P_3 = \{\mathbf{in}(a) \leftarrow \mathbf{out}(b), \mathbf{in}(b) \leftarrow \mathbf{out}(a)\}$ is not safe.

It is clear that safeness is a syntactic condition and it can be checked in linear time. Safe revision programs have several other nice properties, too. They are similar to the properties of Horn logic programs. We gather them all together in the next theorem.

Theorem 5.1 *Let P be a safe revision program. Then, for every database D_I:*

1. *There is a unique D_R such that $\langle D_I, D_R\rangle$ is a P-justified transition.*
2. *The family of sets $\{D \div D_I : D$ is a model of $P\}$ has a least element: the set $D_R \div D_I$, where D_R is a unique P-justified revision of D_I.*
3. *For every well-ordering $\prec$ of P, the result of the sequential revision process for $\prec$ and D_I is a P-justified revision for D_I.*
4. *The unique P-justified revision for D_I can be computed in time proportional to the total size of D_I and P.*

The proof of Theorem 5.1 is based on the following useful lemma.

Lemma 5.2 *Let P be a safe program and let D be any database. Then the necessary change $\langle I, O\rangle$ determined by $P|D$ has the property that $(D \cup I) \setminus O$ is a P-justified revision of D.*

Lemma 5.2 says that in the case of **safe** programs there is no need to guess. The second part of the reduction is still necessary, but then we can compute the unique P-justified revision of D by computing the necessary change. Moreover, it is easy to see that this unique justified revision can be computed in linear time in the total size of an initial database and the program.

MS2 Given a finite revision program P and an element a, decide whether there are databases D_I and D_R such that D_R is a P-justified revision of D_I and $a \in D_I$ if and only if $a \notin D_R$ (informally, decide whether there is a P-justified transition which changes the status of a)

MS3 Given a finite revision program P, an element a and a database D_I, decide whether there is a database D_R such that D_R is a P-justified revision of D_I and $a \in D_R$

MS4 Given a finite revision program P, an element a and a database D_R, decide whether there is a database D_I such that D_R is a P-justified revision of D_I and $a \in D_I$

As before, all these problems are in NP. In addition, it is not hard to see that E1 can be polynomially reduced to each of MS1 and MS2. Hence, both are NP-complete. NP-completeness of MS3 follows from the fact that their restricted versions (when P consists of in-rules only and $D_I = \emptyset$) are equivalent to the problems to decide whether a given element belongs to at least one stable model of a logic program. The problem involving existence of a preimage, MS4, is solvable in polynomial time. Hence, we get the following theorem.

Theorem 4.3 *Problems* MS1 - MS3 *are NP-complete. Problem* MS4 *is in the class P.*

Membership_in_All problems can be regarded as complements of Membership_in_Some problems. One can use Theorem 4.3 to establish their complexity.

We will now present two algorithms for computing all P-justified revisions for a given database D_I. The first of these algorithms, we refer to it as **Guess_and_Check** algorithm, is based directly on the definition of justified revisions and on Theorem 4.1. The idea is to try every possibility for a P-justified revision for D_I. Theorem 4.1 allows us to restrict the search space to subsets of the universe $var(P)$.

Guess_and_Check(P, D_I)
(1) Compute $U = var(P)$
(2) Compute $D'_I = D_I \cap var(P)$
(3) for all subsets D'_R of U repeat:
(4) **if Check_Justified_Revision**(P, D'_I, D'_R)
(5) **then** output $D'_R \cup (D_I \setminus U)$ as a P-justified revision of D_I.

Steps (1) and (2) can be implemented to run in time linear in the size of P and D_I. The loop (3) is executed 2^n times, where n is the size of the universe $var(P)$. Each execution of the loop takes time linear in the size of P and D_I, Hence, the algorithm runs in time $O((m+n)2^n)$, where $n = |var(P)|$ and $m = |D_I|$.

The next algorithm is based on the sequential revision process idea. Namely, it is based on Theorem 3.1 which states that all P-justified revisions of D_I can be found if all possible orderings of rules in P are considered. In the description given below, I stands for all elements inserted until now, O stands for all elements removed until now and D stands for the current database, R consists of all the rules that were already applied and A stands for the rules that can be applied in a current stage. If the algorithm does not generate any output, D_I has no P-justified revisions.

Sequential_Revision_Process(P, D_I)
(1) for all total orderings $\prec$ of P repeat:
(2) $I := \emptyset$
(3) $O := \emptyset$
(4) $D := D_I$
(5) $R := \emptyset$
(6) $A := AR_P(D) \setminus R$
(7) **while** $I \cap O = \emptyset$ **and** $A \neq \emptyset$ **do**
(8) r:= $\prec$-first rule in A

Problems concerned with justified revisions can be grouped into into three broad categories:

Existence: Does a justified revision exist?

Membership_in_some: Does an atom a belong to some justified revision?

Membership_in_all: Does an atom a belong to all justified revisions?

These questions can be further specialized. Let us start with the existence problem. It has three versions:

E1 Given a finite revision program P, decide whether there is a P-justified transition.

E2 Given a finite revision program P and a finite database D_I, decide whether there is a database D_R such that D_R is a P-justified revision of D_I.

E3 Given a finite revision program P and a finite database D_R, decide whether there is a database D_I such that D_R is a P-justified revision of D_I.

Given a finite program P and two finite sets D_I and D_R one can check in linear time whether D_R is a P-justified revision of D_I. Consider the following algorithm.

Check_Justified_Revision(P, D_I, D_R)
(1) Compute the program P_{D_R} (as in Stage 1 of Definition 2.3)
(2) Compute the reduct $P_{D_R}|D_I$ (as in Stage 2 of Definition 2.3)
(3) Encode $P_{D_R}|D_I$ as a Horn program Q (as in Definition 2.2)
(4) Compute the least model M of Q
(5) Decode from M the necessary change (I, O) for $P_{D_R}|D_I$
(6) **if** $I \cap O \neq \emptyset$ **then return**{**false**}
(7) **else if** $D_R \neq D_I \cup I \setminus O$ **then return**{**false**}
(8) **else return**{**true**}

Steps (1), (2), and (6) - (8) correspond to Definition 2.3. Steps (3) - (5) correspond to Definition 2.2. Hence, it is easy to see that algorithm **Check_Justified_Revision** correctly checks whether D_R is a P-justified revision of D_I. Notice also that step (4) can be accomplished in time proportional to the size of Q [DG84]. Hence, the whole algorithm can be implemented to run in time linear in the size of P, D_I and D_R.

By Theorem 4.1 it follows that each of problems E1 - E3 is in NP. Problem E2 is, in fact, NP-complete. It follows from the observation that under the restriction to programs consisting of in-rules only and to the case $D_I = \emptyset$, problem E2 becomes equivalent to the question whether a logic program has a stable model, which is known to be NP-complete [MT91]. Since the satisfiability problem is polynomially reducible to E1, E1 is NP-complete. Problem E3 is simpler. If D_R is not a model for P the answer is NO. Otherwise, D_R is a P-justified revision of itself. Since checking whether D_R is a model of P can be accomplished in time linear in the size of P and D_R, problem E3 can be solved in linear time, too. These observations are summarized in the following theorem.

Theorem 4.2 *Problems* E1 *and* E2 *are NP-complete. Problem* E3 *can be decided in time linear in the size of P and D_R.*

Next, we will consider the Membership_in_some problem. Specifically, we will consider the following versions of this problem:

MS1 Given a finite revision program P and an element a, decide whether there are databases D_I and D_R such that D_R is a P-justified revision of D_I and $a \in D_I$ if and only if $a \in D_R$ (informally, decide whether there is a P-justified transition which does not change the status of a)

and

$$A = AR_P(D) \setminus \{r_{\xi_\gamma} : 1 \leq \gamma < \alpha\}.$$

(The set D describes the database prior to step α in the construction, the set A consists of all rules that are applicable with respect to the database D and have not been applied in the construction until now.) If $A = \emptyset$ then we stop the construction and set $\eta^* = \alpha$. Otherwise, we define

$$\xi_\alpha = \min\{\xi : r_\xi \in A\}.$$

Next, if $head(r_{\xi_\alpha}) = \mathbf{in}(a)$, define

$$I_\alpha = I \cup \{a\}, \quad O_\alpha = O.$$

Otherwise, if $head(r_{\xi_\alpha}) = \mathbf{out}(a)$, define

$$I_\alpha = I, \quad O_\alpha = O \cup \{a\}.$$

If $I_\alpha \cap O_\alpha \neq \emptyset$, define $\eta^* = \alpha + 1$ and stop.

After the construction terminates, define $I_R = \bigcup_{\gamma < \eta^*} I_\gamma$, $O_R = \bigcup_{\gamma < \eta^*} O_\gamma$ and $D_R = (D_I \cup I_R) \setminus O_R$. Note that η^* and the sequences $\{I_\gamma\}_{\gamma < \eta^*}$, $\{O_\gamma\}_{\gamma < \eta^*}$, and $\{\xi_\gamma\}_{1 \leq \gamma < \eta^*}$ depend on the well-ordering $\prec$ of P. We suppressed $\prec$ in the notation in order to simplify it. Let us also observe that if P is finite, all ordinal numbers appearing in the construction are also finite.

The process described above is called the *sequential revision process* for the ordering $\prec$ and a database D_I. Its *result* is a database D_R. In Examples 3.1 and 3.2 we saw that the result of a sequential revision process is not necessarily a P-justified revision of D_I. We will now investigate conditions under which it is so.

A well-ordering of a revision program P is called *a posteriori consistent* for D_I if all the rules of P that were applied in the corresponding sequential revision process are applicable with respect to the resulting database D_R. It is called *sound for a database* D_I if $I_R \cap O_R = \emptyset$. The ordering considered in Example 3.1 is not *a posteriori* consistent for $D_I = \emptyset$, The ordering given in Example 3.2 is not sound for $D_I = \emptyset$.

Theorem 3.1 *Let P be a revision program and let D_I be a database. A database D_R is a P-justified revision of D_I if and only if there exists a well-ordering of P which is* a posteriori *consistent and sound for D_I and such that D_R is the result of the corresponding sequential revision process.*

Theorem 3.1 states that P-justified revisions correspond to a well-motivated class of orderings of the revision program P. It allows us to construct a P-justified revision of D_I by means of a process in which rules are applied sequentially one by one, assuming an *a posteriori* consistent and sound ordering of P can be found.

4 Complexity and algorithms

We will now study the complexity of problems involving justified revisions. For related results see [EG92]. We will also present two algorithms for computing justified revisions given a finite revision program and a finite initial database. We use a certain "localization" result, which says that the status of an element can get changed only if it is mentioned in P. The status of other elements remains unchanged.

Theorem 4.1 (Localization Theorem) *Let P be a revision program. A database D_R is a P-justified revision of a database D_I if and only if*

1. *$D_R \cap var(P)$ is a P-justified revision of $D_I \cap var(P)$, and*
2. *$D_R = (D_R \cap var(P)) \cup (D_I \setminus var(P))$.*

For example, the rule $\mathbf{in}(c) \leftarrow \mathbf{in}(a), \mathbf{out}(b)$ is D-applicable if $D = \{a, b\}$ and it is not D-applicable if $D = \{a, d\}$.

If a rule C is D-applicable then its conclusion can be executed on the database D and, according to the type of the head of C, an atom will be inserted to or deleted from D. Assume that a certain well-ordering $\prec$ of the rules of P is given. Then, the following *sequential revision process* can be considered: in each step select the first rule according to $\prec$ which has not been selected before and which is applicable with respect to the **current** state of the database. Modify the database according to the head of the selected rule. Stop when a selection of a rule is no longer possible. The question that we deal with in this section is: how the results of such revision process relate to P-justified revisions?

Example 3.1 Let $D = \emptyset$ and let P consist of the following two rules:

(1) $\mathbf{in}(c) \leftarrow \mathbf{out}(b)$
(2) $\mathbf{in}(b) \leftarrow \mathbf{in}(c)$.

Let us process the rules in the order they are listed. Rule (1) is applicable with respect to $D = \emptyset$. Hence, the update $\mathbf{in}(c)$ is executed and we get a new database $D_1 = \{c\}$. The second rule is the first D_1-applicable rule not applied yet. Hence, the update $\mathbf{in}(b)$ is executed. Consequently, the next database $D_2 = \{b, c\}$ is obtained. Since there are no other rules left, the process stops. Notice, however, that rule (1) is not D_2-applicable. Hence, the justification for inserting c is lost and D_2 should not be regarded as a revision of D. Observe that D_2 is not a P-justified revision of D, either.

Example 3.1 shows that there are cases when processing rules sequentially does not lead to a P-justified revision. The problem is that some of the rules applied at the beginning of the process may be rendered inapplicable by subsequent updates. But there is yet another source of problems.

Example 3.2 Let $D = \emptyset$ and let P consist of the following three rules:

(1) $\mathbf{in}(c) \leftarrow \mathbf{out}(b)$
(2) $\mathbf{in}(d) \leftarrow \mathbf{in}(a)$
(3) $\mathbf{out}(c) \leftarrow \mathbf{in}(d)$.

Let us process the rules in the order they are listed. After using rule (1) we get a new database: $D_1 = \{c\}$. Then, rule (2) is D_1-applicable and after the update we obtain the database $D_2 = \{c, d\}$. Finally, we apply rule (3) and produce the database $D_3 = \{d\}$. Notice that all the rules applied in the process are D_3-applicable. But D_3 is not a model of the program P and sets of inserted and deleted atoms are not disjoint. Hence, it cannot be regarded as a possible revised version of D. Observe also that, since D_3 is not a model of P it is not a P-justified revision of D.

Example 3.2 shows another case when processing rules of the program according to some ordering does not yield a P-justified revision. It turns out that Examples 3.1 and 3.2 capture all such cases.

We will now formally define the *sequential revision process* and provide a precise formulation of the statement above. The approach we take is similar to our earlier result in which default extensions (and, hence, also stable models of logic programs) are characterized as results of some sequential computation by means of default rules (program clauses) [MT93].

Let D_I be a set of atoms (a database) and let P be a revision program. Both D_I and P may be infinite. Let $\{r_\xi\}_{\xi<\eta}$ be the enumeration of rules in P (here and below, we will use Greek letters to denote ordinals) corresponding to some well-ordering $\prec$ of P. We define an ordinal η^*, a sequence of ordinals $\{\xi_\gamma\}_{1\leq\gamma<\eta^*}$ and two sequences of sets $\{I_\gamma\}_{\gamma<\eta^*}$ and $\{O_\gamma\}_{\gamma<\eta^*}$ as follows. First, we define

$$I_0 = \emptyset, \quad O_0 = \emptyset.$$

Let $\alpha \geq 1$ be on ordinal number. Assume that we have already defined I_γ and O_γ, for $\gamma < \alpha$ and ξ_γ for $1 \leq \gamma < \alpha$. Set

$$I = \bigcup_{\gamma<\alpha} I_\gamma, \quad O = \bigcup_{\gamma<\alpha} O_\gamma, \quad D = (D_I \cup I) \setminus O$$

Theorem 2.1 ([MT94]) *1. Let P be a revision program and let (I,O) be the necessary change determined by P. Then, for every model M of P, $I \subseteq M$ and $O \cap M = \emptyset$.*

2. If a database D satisfies a revision program P then D is a unique P-justified revision of D.

3. Let P be a revision program and let D_I be a database. If a database D_R is a P-justified revision of D_I, then D_R is a model of P.

Revision programming can be viewed as a generalization of logic programming. Given a logic program clause $C = p \leftarrow q_1, \ldots, q_m, \mathbf{not}(s_1), \ldots, \mathbf{not}(s_n)$ we define the revision rule $r(C)$ as

$$\mathbf{in}(p) \leftarrow \mathbf{in}(q_1), \ldots, \mathbf{in}(q_m), \mathbf{out}(s_1), \ldots, \mathbf{out}(s_n). \tag{3}$$

In addition, for a logic program P, we define the corresponding revision program $r(P)$ by

$$r(P) = \{r(C) : C \in P\}. \tag{4}$$

Theorem 2.2 (Stability theorem) *Let P be a logic program. A set of atoms M is a model of P if and only if M is a model of $r(P)$. A set of atoms M is a stable model of P if and only of M is an $r(P)$-justified revision of $\emptyset$.*

One of the reasons for adopting justified revisions to describe database transformations entailed by revision programs is given in the next result. It states, that justified revisions satisfy some minimality condition. Specifically, the change is minimized.

Theorem 2.3 (Minimality theorem) *Let P be a revision program and let D_I be a database. If D_R is a P-justified revision of D_I, then $D_R \div D_I$ ($\div$ stands for the symmetric difference) is minimal in the family $\{D \div D_I : D$ is a model of $P\}$.*

Finally, even a cursory inspection of our construction of justified revisions shows that, unlike in ordinary logic programming, each of the stages of the reduction process is symmetric. That is the same principle is used for positive and negative literals in the body. This symmetry phenomenon actually is more general. We denote by $\overline{D}$ the complement of D i.e. $At \setminus D$. The program P^D called *dual* of P arises from P by simultaneous substitution of **in** for **out** and conversely.

Theorem 2.4 (Duality theorem) *Let P be a revision program and let D_I be a database. Then, D_R is a P-justified revision of D_I if and only if $\overline{D_R}$ is a P^D-justified revision of $\overline{D_I}$.*

Stability, Minimality and Duality Theorems have been proved in [MT94]. All the remaining results in this paper are original and not published or submitted elsewhere.

3 Sequential revision process

Our definition of P-justified revisions has a certain "global" character. It is based on two operators that are applied to programs rather than to individual rules. First of these operators assigns the reduct to a revision program, the other one assigns to the reduct the necessary change it implies. Hence, P-justified revisions of D_I can be viewed as the results of applying all rules of P to D_I "in parallel". In this section, we will present a different description of P-justified revisions. We will show that P-justified revisions of D_I are exactly those databases D_R which can be obtained from D_I by executing all rules of P **one by one** according to some enumeration of the rules in P. This property of the semantics of P-justified revisions is similar to the notion of *serializability* in transaction management.

Let C be a revision rule and let D be a database. If D satisfies the body of the rule C, then C is *applicable* with respect to D (D-*applicable*, for short). Let P be a revision program. We define

$$AR_P(D) = \{C \in P : C \text{ is } D\text{-applicable}\}.$$

respectively) if $p \in D$ ($p \notin D$, respectively), or if there is i, $1 \leq i \leq m$, such that $q_i \notin D$, or if there is i, $1 \leq i \leq n$, such that $s_i \in D$. A database D satisfies a revision program P if D satisfies each rule in P.

In order to apply revision programming to a database D, we need to take for U the set of all ground atomic formulas $p_R(a_1, \ldots, a_k)$. To specify the insertion of $(a_1, \ldots, a_k)$ into a relation R of D we can use an in-rule: $\mathbf{in}(p_R(a_1, \ldots, a_k)) \leftarrow$. Similarly, the deletion can be specified as the out-rule $\mathbf{out}(p_R(a_1, \ldots, a_k)) \leftarrow$. More generally, rules of type (1) describe integrity constraints of the form: if tuples $q_1, \ldots, q_m$ are in a database and tuples $s_1, \ldots, s_n$ are not in the database, then tuple p is in the database. Rules of type (2) have a similar interpretation, except that they stipulate that tuple p be **not in** the database.

We use revision programs as means to specify integrity constraints. We assume some initial state D_I of a database. If D_I satisfies all constraints in a revision program P, no change in D_I is necessary. If, however, D_I does not satisfy P, we use P as an input-output device to **enforce** on D_I the constraints it represents. That is, we produce several (possibly none) databases D_R each of which satisfies the constraints of P. Moreover, we do so in such a way that each change (insertion, deletion) necessary to transform D_I into D_R is **justified**.

A detailed discussion of motivations for our approach to revisions as well as several examples is given in [MT94].

We will now present a construction introduced in [MT94], which describes a mechanism for enforcing rules in revision programs. It is based on the notion of *necessary change* — the change that is entailed by a program alone, that is, without references to any databases.

Definition 2.2 Let P be a revision program. Let Q be the Horn program obtained from P by treating each literal in P as a separate propositional variable. Let M be the least model of Q. The *necessary change* for P (or, *determined by* P) is defined as the pair (I, O), where $I = \{a : a \text{ is an atom and } \mathbf{in}(a) \in M\}$ and $O = \{a : a \text{ is an atom and } \mathbf{out}(a) \in M\}$. If the least model M of Q does not contain any pair $\mathbf{in}(a)$, $\mathbf{out}(a)$, that is, if $I \cap O = \emptyset$, then P is called *coherent*.

In general, necessary change is not sufficient to compute a revised database. The initial database as well as a tentative final one (used in a way reminiscent of the construction of stable models [GL88]) must be taken into account.

Definition 2.3 ([MT94]) Let P be a revision program and let D_I and D_R be two databases.

1. The reduct of P with respect to (D_I, D_R) is defined in two stages:

   Stage 1: Eliminate from P every rule of type (1) or (2) such that $q_i \notin D_R$, for some i, $1 \leq i \leq m$, or $s_j \in D_R$, for some j, $1 \leq j \leq n$. The resulting program is denoted by P_{D_R}.

   Stage 2: From the body of each rule that remains after Stage 1 eliminate each $\mathbf{in}(a)$ such that $a \in D_I$ and each $\mathbf{out}(a)$ such that $a \notin D_I$.

2. The program resulting from P after both stages are executed is called the *reduct of P with respect to* (D_I, D_R) and is denoted by $P_{D_R}|D_I$.

3. Let (I, O) be the necessary change determined by $P_{D_R}|D_I$. If $I \cap O = \emptyset$ (that is, if $P_{D_R}|D_I$ is coherent) and $D_R = (D_I \cup I) \setminus O$, then D_R is called a *P-justified revision of D_I* and the pair $\langle D_I, D_R \rangle$ is called a *P-justified transition*. We will write $D_I \xrightarrow{P} D_R$ in such case.

Informally, in order to treat D_R as a P-justified revision of D_I, we need to have justifications for every deletion and insertion that are needed to transform D_I into D_R. The justification must be valid **after** the revision. Hence, only rules of $P|D_R$ can be used as justifications. In addition, the initial status D_I of the database must be taken into account (Stage 2). If the necessary change entailed by the resulting program $P_{D_R}|D_I$ converts D_I into D_R, D_R is a P-justified revision of D_I.

The following result describes the most fundamental properties of satisfaction and necessary change.

analogies with logic programming. In this paper, we discuss applications of revision programming to study updates and integrity constraints in databases and investigate results on algorithmic aspects of revision programming.

In the next section we recall basic definitions and results from [MT94]. In Section 3, we show that the semantics of justified revisions has a certain "serializability" property. Namely, justified revisions are exactly the results of the *sequential revision process* in which rules are executed according to a certain order. Next, we study the complexity of several decision problems involving justified revisions and we present algorithms to compute them. These algorithms are computationally complex. In the last section we introduce a class of programs, called *safe* programs, which have the property that for an arbitrary database there is a unique justified revision. Finite propositional safe programs can be recognized in linear time. In addition, for every initial database the unique justified revision can be computed in linear time (in the total size of the program and the initial database). Safe programs play in revision programming the role analogous to that played by Horn programs in logic programming. Extending the analogy, the class of safe programs is generalized to the class of *stratified* revision programs. Two key properties of safe programs are preserved. First, stratified programs can be recognized in polynomial time. Second, computing justified revisions for stratified revision programs is linear (in the total size of the program and the initial database). We also identify a class of sets studied in classical recursion theory associated with justified revisions of recursive databases determined by recursive safe program.

From the database perspective the classes of safe and stratified revision programs are especially important as they have good computational properties and they always determine a unique justified revision.

Although there is an immense literature devoted to updates in databases, the roots of our work are in logic programming and knowledge representation. The technique developed here is based on the work of Reiter [Rei80] and Gelfond and Lifschitz [GL88]. However, motivations as well as some key ideas come from database theory. First, we treat programs as input-output devices, as it is the case in DATALOG [Ull88]. Second, our language is similar to that used by Abiteboul and Vianu [AV90, AV91] in their work on extensions of DATALOG admitting deletions. Stratified programs were studied (in a different setting) by Manchanda and Warren [MW88], who assigned to them a Kripke-style semantics.

2 Preliminaries

In this section we present basic concepts of revision programming introduced in [MT94]. We will encode a database D as a theory consisting of ground atomic formulas of a certain first-order language. Namely, for each k-ary relation R in the database schema we have a predicate symbol p_R whose intended meaning is: $p_R(x_1, \ldots, x_k)$ if and only if $(x_1, \ldots, x_k) \in R$. Then, each tuple $(a_1, \ldots, a_k)$ of R is represented by the ground atom $p_R(a_1, \ldots, a_k)$.

Definition 2.1 Let U be a denumerable set. We call its elements *atoms*. A *revision in-rule* or, simply, an *in-rule*, is any expression of the form

$$\mathbf{in}(p) \leftarrow \mathbf{in}(q_1), \ldots, \mathbf{in}(q_m), \mathbf{out}(s_1), \ldots, \mathbf{out}(s_n), \tag{1}$$

where p, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are all in U. A *revision out-rule* or, simply, an *out-rule*, is any expression of the form

$$\mathbf{out}(p) \leftarrow \mathbf{in}(q_1), \ldots, \mathbf{in}(q_m), \mathbf{out}(s_1), \ldots, \mathbf{out}(s_n), \tag{2}$$

where p, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are all in U. All in- and out-rules are called *rules*. Expressions $\mathbf{in}(a)$ and $\mathbf{out}(a)$ are called *literals*. Literals $\mathbf{in}(p)$ and $\mathbf{out}(p)$ are called the heads of the rules (1) and (2). The head of a rule r is denoted by $head(r)$. A collection of rules is called a *revision program* or, simply, a *program*. The set of all literals appearing in a program (as the heads of the rules in a program) is denoted by $var(P)$ ($head(P)$, respectively). A database D *satisfies* a rule (1) (rule (2),

1 Introduction

We study a formalism for stating and enforcing integrity constraints in databases. Integrity constraints can be described in the language of first-order logic, which is quite expressive and allows us to formulate a wide range of constraints. However, there is a problem: if no restrictions on the syntax of integrity constraints are imposed, the classical semantics of the language of first-order logic does not entail a mechanism for *enforcing* them. In this paper, we describe a version of the first-order logic language consisting of rules rather than formulas. In that we follow the approach of logic programming [vEK76, Apt90] but our class of rules is much more general. Our system allows us to specify integrity constraints in a declarative fashion similarly as in the case of the standard first-order language (or logic programming). What is more important, it provides an imperative interpretation to these integrity constraints and, consequently, a mechanism to enforce them, given the initial state of a database. In this paper we will study basic properties of our formalism and we will argue that it is a convenient tool for describing updates on databases and integrity constraints which databases must satisfy.

There are fragments of first-order logic that can be given an imperative interpretation. For example, as first pointed by Van Emden and Kowalski [vEK76], definite Horn programs have least Herbrand models and these models can be described in a procedural fashion.

This match between declarative first-order logic semantics and procedural treatment of rules in logic programming disappears for wider classes of theories. Logic programming writes clauses in such a way as to underline their imperative, computational, character. Namely, a logic program clause is any expression of the form

$$\gamma \leftarrow \alpha_1, \ldots, \alpha_k, \neg\beta_1, \ldots, \neg\beta_m,$$

where α_i, β_i and γ are propositional atoms (the restriction to the propositional case is not needed. We adopt it for the sake of simplicity of discussion). It is interpreted as a mechanism for computing γ after α_i, $1 \leq i \leq k$ have been computed and after it has been established that β_i, $1 \leq i \leq m$, cannot be computed. This last part is somewhat controversial. Several proposals were made to formalize the requirement "β_i cannot be computed". They led to different semantics for logic programs [Cla78, Fit85, VRS91]. One of the most successful is the semantics of stable models [GL88]. The key idea behind the stable model semantics is to select a tentative model M, regard atoms not in M as those that cannot be computed, and use this information in the van Emden-Kowalski computation process. If M is what is produced as the result, M is a *stable* model. Hence, $\neg\beta$ is interpreted as **absence from a tentative model**, while positive literals have to be computed.

We will consider updates (insertions and **deletions**), and integrity constraints which, in the case of some data being present in a database and some data being absent from the database, require that some other data be present in (absent from) the database. The formalism of logic programs with stable semantics is not expressive enough to be directly employed as a specification language for database updates and integrity constraints. Logic program clauses can only compute new atoms and, thus, can model rules which require items be inserted into databases. But they cannot model deletions. No logic program clause can express an imperative rule **(under some conditions) delete a record**, since negative literals are not allowed in the head. Second, less troublesome limitation, of logic programming is that logic programs compute from the empty set of atoms while updates and integrity constraints must be enforced on arbitrary databases. DATALOG [Ull88] overcomes this difficulty.

In this paper, we present a system which allows us to form rules with negative literals in the head. We call it *revision programming*. For revision programs, we describe a semantics of justified revisions. This semantics describes, for a given set of atoms, all "justified" ways in which to modify it in order to satisfy the constraints specified by a revision program. Consequently, our formalism can be used to formulate and process integrity constraints and updates.

Revision programming extends logic programming and the notion of a justified revision generalizes that of a stable model. It is different, though, from logic programming with classical negation [GL90].

We have introduced revision programs and justified revisions in [MT94]. We have studied there basic properties of revision programming, mentioned a simple application to belief revision and considered

Revision programming, database updates and integrity constraints

Victor W. Marek
Mirosław Truszczyński
Department of Computer Science
University of Kentucky
Lexington, KY 40506-0027
{marek,mirek}@ms.uky.edu

Fax: (606)-323-1971
Phone: (606)-257-3961

Keywords: revision programming, integrity constraints, updates, justified transitions, safe programs, stratified programs

Abstract

We investigate revision programming, a logic-based mechanism for description of changes in databases. We show that revisions justified by an initial database and a revision program can be computed by a sequential execution of the rules of the program (with subsequent check of the applicability of the rules). In general, a program may determine none, exactly one or many justified revisions of a given initial database. We exhibit two classes of programs, *safe* and *stratified*, with the property that for every initial database a **unique** justified revision exists. We study the complexity of basic problems associated with justified revisions. Although the existence problems are NP-complete, for safe and stratified programs justified revisions can be computed in polynomial time.