Update Policies

Abhijeet Mohapatra, Sudhir Agarwal, and Michael Genesereth

Computer Science Department, Stanford University, USA
{abhijeet,sudhir,genesereth}@cs.stanford.edu

Abstract. Incomplete transaction can be supported in databases by enabling administrators to express update policies that complete incomplete transactions. In this paper, we propose a language for expressing such update policies. We show that the problem of verifying whether or not a policy is sound and complete is undecidable in general. We identify decidable instances of this decision problem, and for such instances, present an algorithm that uses resolution to check whether or not a supplied policy is sound and complete.

1 Introduction

Many software systems use databases to model the state of the world. Such databases consist of base relations, derived relations (a.k.a. views) and constraints. Transactions on a database change the state of the database.

For ease of use it is desirable to allow users to specify incomplete transactions. In some cases database constraints contain sufficient information for completing an incomplete transaction unambiguously. In other cases there may exist multiple ways to complete an incomplete transaction.

Traditional approaches typically reject incomplete transactions. The approach presented in [Orm01] can compute all necessary and sufficient conditions for completing an incomplete transaction. However, transaction repair approaches including the one presented in [Orm01] do not incorporate administrators' preferred strategies for transaction completion. The approach of Courteous Logic Programs presented in [GLC99] enables expressing prioritized conflict handling during query evaluation in one database state whereas a transaction completion strategy involves two database states.

In this paper, we investigated the problem of expressing update policies for updating databases. We give formal definition of minimal inclusive update policies and present an expressive and declarative update policy language to enable administrators to defining their update policies. While update policies help defining the desired database behavior in ambiguous situations in a powerful and flexible way, an update policy that can lead to an illegal database state is often not desirable. In order to support administrators in authoring update policies that are consistent with the database constraints, we also present a resolution-based technique to check update policies for their soundness and completeness wrt. database constraints.

2 Problem Definition

A database consists of relations. An instance of a database is a subset of the set of all ground atoms that can be formed using the base relation constants and symbols from the domain. A database instance is updated through *transactions*.

Definition 1 (Transactions). *A transaction on a database instance is a tuple* $\langle T_i, T_d \rangle$*, where* T_i *denotes the set of atoms that are inserted into the database instance,* T_d *denotes the set of atoms that are deleted from the instance. In addition,* $T_i \cap T_d = \emptyset$*,* $T_i \cap D = \emptyset$*, and* $T_d \subseteq D$*.*

A database may also contain *constraints* which are the set of legal database instances.

Definition 2 (Legal Transactions). *A transaction* $\langle T_i, T_d \rangle$ *on a database instance* D *is legal if the instance* $(D \setminus T_d) \cup T_i$ *is legal.*

Example 1. Consider a database that contains a single unary base relation p. Suppose the set of all ground atoms in the database is $\{p(a), p(b)\}$, and the database instance $\{p(b)\}$ illegal. As a result, $\langle \{p(b)\}, \{\} \rangle$, $\langle \{p(b)\}, \{p(a)\} \rangle$ and $\langle \{\}, \{p(a)\} \rangle$ are illegal transactions with respect to the database instances $\{\}$, $\{p(a)\}$, and $\{p(a), p(b)\}$ respectively.

The traditional approach for enforcing constraints is to reject illegal transactions. However, there are cases where it is desirable for users to specify only a subset of the *intended* transaction. For some of these cases, the specified input is a *complete* specification of the transaction. In other words, there exists a unique transaction that is a superset of the input and is legal.

Example 2. Consider a modeling of the blocks-world with two blocks a and b as a database with a binary relation *on* and two unary relations *table* and *clear*. These relations characterize whether or not a block is placed directly on another block, a block is on the ground, and a block has no other blocks on it.

Suppose that the blocks a and b are initially placed on the ground.Therefore, the database instance contains the atoms *table*(a), *table*(b), *clear*(a), and *clear*(b). Suppose, we specify that in the next state of the blocks-world, a is placed on b.

We note that, this specification asks for the atom *on*(a, b) to be inserted in to the model, but does not explicitly specify whether or not the atoms *table*(a) and *clear*(b) are deleted from the model. However, these additional effects can be derived from the constraints on the blocks world.

In other scenarios, the input is an *incomplete* specification of the transaction i.e., there exist multiple supersets of the input that are legal transactions. For example, adding a third block, say c, to the model of the blocks world in Example 2 results in such a scenario. Suppose that analogous to blocks a and b, c was initially placed on the ground. In this setting, the specification that a is placed on b in the next state does not clarify whether c is on the ground, or on a in the next state.

To enable system administrators to deal with scenarios where users may specify a subset of the intended transaction, we present an abstract framework that supports the authoring of strategies by administrators construct *completed* transactions from a subset thereof. In the following, we define the concepts of our framework.

Definition 3 (Transaction Requests). *A transaction request on a database instance is a tuple $\langle R_i, R_d \rangle$, where R_i denotes a set of atoms that are requested to be inserted into the database instance, R_d denotes the set of atoms that are requested to be deleted from the instance. In addition, $R_i \cap R_d = \emptyset$, $R_i \cap D = \emptyset$, and $R_d \subseteq D$.*

Proposition 1. *For every database instance D, a legal transaction $\langle T_i, T_d \rangle$ on D, sets A and B such that $A \subseteq T_i$, $B \subseteq T_d$, there exists sets C and D where $A \subseteq C$, $B \subseteq D$, and $\langle C, D \rangle$ is a legal transaction.*

Definition 4 (Update Policies). *An update policy over a database is a function that takes as input a legal database instance D and a transaction request $\langle R_i, R_d \rangle$ on D, and outputs a legal transaction $\langle T_i, T_d \rangle$ on D.*

A policy is inclusive if for every input: D, $\langle R_i, R_d \rangle$, the policy outputs $\langle T_i, T_d \rangle$ such that $R_i \subseteq T_i$ and $R_d \subseteq T_d$.

An inclusive policy is minimal if for every input: D, $\langle R_i, R_d \rangle$, the policy outputs $\langle T_i, T_d \rangle$ such that for all T'_i and T'_d, where $T'_i \subsetneq T_i$ and $T'_d \subseteq T_d$, or $T'_i \subseteq T_i$ and $T'_d \subsetneq T_d$, $\langle T'_i, T'_d \rangle$ is not a legal transaction on D.

In our abstract framework, an inclusive policy corresponds to an administrator's strategy for augmenting transaction requests, which may potentially be under-specified, to construct a *completed* transaction. It may also be desirable for an update policy to allow users to completely specify a transaction in their request. In other words, every legal state in the database should be reachable from every other legal state. However, an arbitrary inclusive policy may violate this condition as illustrated in the following example.

Example 3. Consider the setting in Example 1, and an update policy over the database with the following properties.

- The update policy results in the transaction $\langle \{p(a), p(b)\}, \{\} \rangle$ for the transaction requests $\langle \{p(a)\}, \{\} \rangle$ and $\langle \{p(b)\}, \{\} \rangle$ on the empty database instance.
- The update policy results in the transaction $\langle \{\}, \{p(a), p(b)\} \rangle$ for the transaction requests $\langle \{\}, \{p(a)\} \rangle$ and $\langle \{\}, \{p(b)\} \rangle$ on the database instance $\{p(a), p(b)\}$.

We note that an update policy that results in the above transactions satisfies Definition 4. However, as a result of such an update policy, the instance $\{p(a)\}$ is not reachable from the instances $\{\}$ or $\{p(a), p(b)\}$ through any transaction.

Proposition 2. *For every pair of legal database instances D and D', every minimal update policy maps a transaction request $\langle D' \setminus D, D \setminus D' \rangle$ on D to $\langle D' \setminus D, D \setminus D' \rangle$.*

In order, to ensure that update policies do not prevent users from completely specifying a transaction in their request, it may be desirable that update policies be minimal. At first, the minimality requirement on inclusive policies may seem stringent. However, we show that there always is a minimal policy that ensures the simulation of a supplied legal transaction through a sequence of singleton transaction requests. Such a property is desirable when there are limits on the size of the transaction e.g. when submitting a batch API request to the Facebook API[1] or the Twitter API[2].

Theorem 1 (Unit Serializability). *For every database instance D, and a legal transaction $\langle T_i, T_d \rangle$ on D, there exists a minimal policy such that all of the following conditions are satisfied.*

- *S_T is the set of singleton transactions $\{\langle\{A\},\{\}\rangle \mid A \in T_i\} \cup \{\langle\{\},\{A\}\rangle \mid A \in T_d\}$.*
- *There exists an integer k such that $1 \leq k \leq |S_T|$, and a sequence of k distinct transactions from S_T such that applying these transactions in a sequence over D results in a database instance that is identical to the one obtained by applying $\langle T_i, T_d \rangle$ on D.*

Proof. We prove the above result using induction on size of $T_i \cup T_d$. The *base case* trivially holds i.e. when $|T_i| = 1$ or $|T_d| = 1$. In the *induction hypothesis*, we assume that for $|T_i \cup T_d| \leq k$, the theorem holds.

For the *induction step*, we assume that $|T_i \cup T_d| = k + 1$. Consider a pair of sets A and B, such that $A \subseteq T_i$ and $B \subsetneq T_d$, or $A \subsetneq T_i$ and $B \subseteq T_d$. It is either the case that (a) there exists some A and B such that transaction $\langle A, B \rangle$ is legal, or (b) for every A and B $\langle A, B \rangle$ is an illegal transaction. If there exists A and B such that $\langle A, B \rangle$ is legal, then it follows that $\langle T_i \backslash A, T_d \backslash B \rangle$ is legal. Therefore, the theorem holds by applying the induction hypothesis. In case (b), we construct a minimal policy by assigning $\langle T_i, T_d \rangle$ to some singleton sub-transaction.

3 Update Policy Language

In this section we present our update policy language which enables administrators to expressively specify update policies. Our update policy language is a Logic Program. So, we first give a brief overview of Logic Programs. Then, we present the syntax of our update policy language and discuss its expressiveness.

3.1 Logic Programs

We encode each instance of a relationship in the form of a *sentence* consisting of a *relation constant* (begins with a small letter) representing the relationship and some *terms* representing the objects involved in the instance.

[1] `https://developers.facebook.com/docs/graph-api/making-multiple-requests`
[2] `https://dev.twitter.com/tags/bulk-operations`

A term is defined as an object constant (begins with a small letter), a variable (begins with a capital letter), or a functional term, i.e. an expression consisting of an n-ary function constant and n simpler terms. An atom is an expression formed from an n-ary relation constant and n terms. An atom is *ground* if it has no variables.

A *literal* is either an atom or a negation of an atom (i.e. an expression stating that the atom is false). A simple atom is called a *positive* literal, The negation of an atom is called a *negative* literal. In what follows, we write negative literals using the negation sign '$\neg$', which represents *negation-as-failure*. For example, if $p(a, b)$ is an atom, then $\neg p(a, b)$ denotes the negation of this atom.

A *rule* is an expression consisting of a distinguished atom, called the head and a conjunction of zero or more literals, called the *body*. The literals in the body are called *subgoals*. In what follows, we write rules as in the example

$$r(X, Y) \text{ :- } p(X, Y), \neg q(Y).$$

Here, $r(X, Y)$ is the head, $p(X, Y), \neg q(Y)$ is the body; and $p(X, Y)$ and $\neg q(Y)$ are subgoals.

Semantically, a rule states that the conclusion of the rule is true whenever the conditions are true. For example, the rule above states that r is true of any object X and any object Y if p is true of X and Y and q is not true of Y. For example, if we know $p(a, b)$ and we know that $q(b)$ is false, then, using this rule, we can conclude that $r(a, b)$ must be true.

A Logic program is a finite set of ground atoms and rules, and can be evaluated in top-down or bottom-up fashion [Ull89] in time that is polynomial in the size of the program.

3.2 Constraints

We represent constraints by specifying the set of illegal database instances. A constraint is a rule of the form $\bot$:- ϕ where ϕ is conjunction of literals. A database instance D satisfies a constraint $\bot$:- ϕ if and only if $D \not\models \phi$. A database instance D satisfies a set of constraints C if and only if D satisfies every constraint in C.

3.3 Update Policy Language

We propose to model update policies as a Logic Program with standard negation and four special operators δ^+, δ^-, Δ^+ and Δ^-. Each of these four special operators take an atom as argument.

In order to encode update policies, we use δ^+ and δ^- to denote R_i and R_d of a requested transaction R respectively. In addition, we use Δ^+ and Δ^- to denote the T_i and T_d of a transaction T respectively. We require that δ^+ and δ^- must not appear in the head of a rule, and Δ^+ and Δ^- must not appear in the body of a rule. A stratified logic program that defines Δ^+ and Δ^- in terms of δ^+, δ^- and the current database instance essentially defines an update policy (see Definition 4).

Example 4. Consider Example 1. The constraint that the database instance $\{p(b)\}$ is illegal can be enforced by the following minimal update policy:

$$\begin{aligned}
&\Delta^{+}p(a) \text{ :- } \delta^{+}p(b),\ \neg p(a).\\
&\Delta^{+}p(b) \text{ :- } \delta^{+}p(b),\ \neg p(a).\\
&\Delta^{+}p(b) \text{ :- } \delta^{+}p(b),\ p(a).\\
&\Delta^{+}p(a) \text{ :- } \delta^{+}p(a).\\
&\Delta^{-}p(a) \text{ :- } \delta^{-}p(a),\ \neg p(b).\\
&\Delta^{-}p(a) \text{ :- } \delta^{-}p(a),\ p(b).\\
&\Delta^{-}p(b) \text{ :- } \delta^{-}p(a),\ p(b).\\
&\Delta^{-}p(b) \text{ :- } \delta^{-}p(b).
\end{aligned}$$

The policy ensures that when $p(b)$ is requested to be inserted then the next database instance contains both $p(a)$ and $p(b)$. The policy also ensures that when $p(a)$ is requested to be deleted, then $p(b)$ is also deleted if $p(b)$ is present. Requests for insertion of $p(a)$ and deletion of $p(b)$ do not require any modification as they cannot lead to an inconsistent database instance.

3.4 Expressiveness of Update Policy Language

As mentioned above, an update policy defines a function from the set of database instances and the set of transaction requests to the set of transactions. If the set of database instances is finite then the set of transaction requests as well as the set of transactions are also finite. Therefore, there are only finitely many update policies. In this case, our update policy language can model all the functions since we can simply enumerate all the functions.

In case there are countably infinite database instances, there are also countably infinite transaction requests and countably infinite transactions. In this case, there are uncountably infinite possible update policies. Therefore, our update policy language can not model all possible update policies.

However, our update policy language is Turing Complete, i.e., it can model all recursively enumerable functions. Functions that our update policy language can not model can not be modeled by any other programming language either.

Comparison with ECA-Rules Commercial database systems use an approach that is similar to ours in enforcing constraints through *triggers*. Triggers are event-condition-action (ECA) rules [WC96] that perform some actions such as applying additional updates or raising exceptions, when an update event occurs on a database instance, and the requested update and the database instance satisfy the conditions of the trigger.

Our update policy language is *strictly more expressive* than the language used in triggers. Any policy that can be implemented with ECA rules can be

implemented with our update policy language as well, and there are policies that can be modeled with our update policy language but cannot be implemented with ECA rules.

An ECA rule (E, C, A) has the meaning that if event E occurs and the condition C is true, then do action A. An ECA-rule can be modeled with out update policy language by modeling an event as a transaction request, a condition as a conjunction of literals, and an action as a transaction.

Now, let us consider a case that our update policy language can implement but triggers cannot. For instance, triggers *cannot* implement the following update policy due to their inability to distinguish between requested updates and triggered updates.

$$\Delta^+ q(X) \text{ :- } \delta^+ p(X).$$
$$\Delta^+ r(X) \text{ :- } \delta^+ q(X).$$

In the trigger-based approach there is no way to stop $r(X)$ from being inserted when $p(X)$ is requested to be inserted except by introducing new relations, i.e. changing the database schema.

Comparison with STRIPS Action Representation The STRIPS [FN71] representation for an action consists of: (a) *preconditions*, a list of atoms that need to be true for the action to occur, (b) a *delete list*, a list of those primitive relations no longer true after the action, and (c) an *add list*, a list of the primitive relations made true by the action.

A STRIPS action can be specified as an update policy. Such an update policy contains one rule for each element in the add and delete lists with each rule having the same body (conjunction of the precondition of the action and all the elements in the add and delete lists), and the head of equals to Δ^+ or Δ^- (depending on if the element in the add list or in the delete list). Since the body of all the rules is the same for an action, either all or none of the changes specified in their head are applied depending on whether the body is true or not. This behavior corresponds to the intended semantics of a STRIPS action.

Example 5. Consider the underspecified variant of the block-world problem introduced in Example 2. Below an update policy for serving a request for moving a block. Assume a unary relation *block* to represent the set of all blocks.

$$\Delta^+ on(X, Y) \text{ :- } \delta^+ on(X, Y),\ clear(X),\ clear(Y).$$
$$\Delta^- on(X, U) \text{ :- } \delta^+ on(X, Y),\ clear(X),\ clear(Y),\ on(X, U).$$
$$\Delta^- clear(Y) \text{ :- } \delta^+ on(X, Y),\ clear(X),\ clear(Y).$$
$$\Delta^+ table(X) \text{ :- } \delta^+ table(X),\ clear(X).$$
$$\Delta^- on(X, W) \text{ :- } \delta^+ table(X),\ clear(X),\ on(X, W).$$

The update policy ensures that when it is requested to move a block to another block then (1) updates occur only when there is no block over the block to be moved and there is no block over the destination block. (2) when updates occur then the block's new position is inserted and its old position is deleted. Analogous but simpler is the case when it is requested to move a block to ground.

Comparison with a STRIPS Extension for Joint-Actions An extension of STRIPS action representation by joint actions for generating concurrent non-linear plans is presented in [BB01]. The formalism for representing joint-actions is obtained by adding a possibly empty *concurrent list* to each action. A concurrent list specifies which actions actions can co-occur or cannot co-occur with the given action in order to produce the described effect.

Such joint actions can be seen and modeled as transaction requests with our update policy language as our update policy language allow multiple δ^+ and δ^- literals in the body of an update policy rule.

In the formalism presented in [BB01], the concurrency lists and the effect of concurrent execution of a and b needs to described twice, once each for action a and action b. The concurrency list of a contains b and the concurrency list of b contains a and analogously the effects. With our update policy language we need to define the effect of concurrent execution of a and b only once.

In addition to and despite the simpler syntax of our update policy language, our update policy language is more expressive than the formalism presented in [BB01]. With our update policy language we can also define effects that are recursively computed from the current database instance and the requested transaction. The formalism presented in [BB01] cannot model such effect. For example the following update policy cannot be modeled with the formalism presented in [BB01].

$$
\begin{aligned}
&v(X,Y) \text{ :- } \delta^+ p(X,Y).\\
&v(X,Z) \text{ :- } \delta^+ p(X,Y),\ v(Y,Z).\\
&\Delta^+ p(X,Y) \text{ :- } v(X,Y).
\end{aligned}
$$

Another interesting difference between the formalism presented in [BB01] and our update policy language concerns multiple occurrences of the same action. While the former does not allow multiple concurrent actions by the same agent our update policy language can deal with multiple occurrences of an action (or more precisely, its equivalent in our update policy language).

4 Verification of Update Policies

In this section, we consider the problem of verifying whether or not a given update policy enforces a given set of constraints. We formally define the verification problem as follows.

Definition 5. *(Decision Problem) Given an update policy P, and a set of static constraints Λ, the VERIFY-POLICY problem decides whether or not application of P on legal database instance D results in a legal transaction $\langle T_i, Td\rangle$ for every transaction request $\langle R_i, R_d\rangle$ i.e. if $D \cup P \cup \langle R_i, R_d\rangle \models \langle T_i, T_d\rangle$, and $D \cup \Lambda \nvdash \bot$, then $D \setminus T_d \cup T_i \cup \Lambda \nvdash \bot$.*

Case 1 (Finite Herbrand Base): First, we consider the case where the *Herbrand base* of the supplied database is finite, and is specified as a part of the input. In this case, a VERIFY-POLICY instance $\langle \Lambda, P\rangle$ can be *decided* as follows. Let B_D denote the Herbrand base of the database. Let L_D denote the set of legal database instances. In our case, a subset D of B_D is in L_D *iff* $D \cup \Lambda \nvdash \bot$. Let D_R be the set of transaction requests $\{\langle R_i, R_d\rangle \mid R_i, R_d \subseteq B_d \wedge R_i \cap R_d = \emptyset\}$. For every $D \in L_D$ and transaction request $\langle R_i, R_d\rangle$ in D_R, we compute the completion $\langle T_i, T_d\rangle$ using the supplied policy P, and the next state of the database $D' = D \setminus T_d \cup T_i$. If $D' \cup \Lambda \vdash \bot$ for some $D \in L_D$ and $\langle R_i, R_d\rangle$ in D_R, then we output *No* as the answer of the VERIFY-POLICY instance $\langle \Lambda, P\rangle$. Otherwise, we output *Yes*.

Case 2 (Non-recursive rules): Instead of exhaustively enumerating the set of all possible legal database instances and legal transactions, we can also decide VERIFY-POLICY instance $\langle \Lambda, P\rangle$ using *resolution* as follows. In this case, we lift the assumptions that the Herbrand base of the supplied database is finite, and s is supplied as input.

We initialize a set of clauses C to be empty.

1. **Assert Legality for Current Instance**: We add to C the clauses in $\neg(\exists \bar{X_1}.\phi_1(\bar{X_1}) \wedge \exists \bar{X_2}.\phi_2(\bar{X_2}) \ldots \wedge \exists \bar{X_k}.\phi_k(\bar{X_k}))$.
2. **Convert Delta and Policy Rules into Clausal Form**: For every constraint in Λ, we generate the *delta* rules [Orm01], which serve as the necessary and sufficient conditions for determining the legality of a transaction on an instance of the supplied database. Let Δ_c denote the set of all delta rules where $\Delta^+\bot$ appears in the head of the rule. For every delta rule in Δ_c of the form $\Delta^+\bot$:- $\phi(\bar{X})$, we add the clause $\forall \bar{X}.\, \phi(\bar{X}) \implies \Delta^+\bot$ to C. Suppose that that there are m delta rules of the form $\Delta^+\bot$:- $\phi_i(\bar{X_i})$ where $i \in [1, m]$. We add to C the clauses in $\Delta^+\bot \implies \bigvee_i (\exists \bar{X_i}.\, \phi_i(\bar{X_i}))$. We convert the rules in P into clausal form in a similar manner, and add the generated clauses to C.
3. **Add Legality for Transaction and Transaction requests** For every base relation r in the database we convert the sentences: (a) $\neg(\Delta^- r(\bar{X}) \wedge \Delta^+ r(\bar{X}))$, (b) $\Delta^- r(\bar{X}) \implies r(\bar{X})$, (c) $\delta^- r(\bar{X}) \implies r(\bar{X})$, (d) $\Delta^+ r(\bar{X}) \implies \neg r(\bar{X})$, and (e) $\delta^+ r(\bar{X}) \implies \neg r(\bar{X})$ into clausal form, and add the generated clauses to C.
4. **Assert Negation of Undefined Differentials**: For every differential relation $\Delta^a r$ where $a \in \{+, -\}$ and $\Delta^a r$ does not appear in the head of any rule in P, we add the clause $\{\neg \Delta^a r(\bar{X})\}$ to C.
5. **Resolution Step**: We add the goal clause $\{\Delta^+\bot\}$ to C and perform resolution. If the empty clause is obtained, then the supplied update policy satisfies

the constraints, and we output *Yes* as the answer to the VERIFY-POLICY instance $\langle \Lambda, P \rangle$. Otherwise, we output *No*.

The above resolution technique always terminates for Case 1. In addition, the above procedure can also be use to decide VERIFY-POLICY in cases where the view definitions in the supplied database are *non-recursive*. In such cases, the resolution is guaranteed to terminate. We illustrate our resolution technique in the following example.

Example 6. Consider a database that contains the following static constraint $\bot$:- $p(b)$. Suppose we want to verify whether or not the following update policy P satisfies the above constraint.

$$\Delta^- p(b) \text{ :- } \Delta^+ p(X).$$

We initialize C to be the empty set of clauses, and add to C the following clauses.

$$\begin{aligned}
c_1 &: \{\neg p(b)\} && \text{(step 1)}\\
c_2 &: \{\neg \Delta^+ p(b), \Delta^+ \bot\} && \text{(step 2)}\\
c_3 &: \{\Delta^+ p(b), \neg \Delta^+ \bot\} && \text{(step 2)}\\
c_4 &: \{\neg \Delta^- p(b), \Delta^+ p(k)\} && \text{(step 2)}\\
c_5 &: \{\Delta^- p(b), \neg \Delta^+ p(X)\} && \text{(step 2)}\\
c_6 &: \{\neg \Delta^+ p(X), \neg \Delta^- p(X)\} && \text{(step 3)}\\
c_7 &: \{\neg \Delta^+ p(X), \neg p(X)\} && \text{(step 3)}\\
c_8 &: \{\neg \delta^+ p(X), \neg p(X)\} && \text{(step 3)}\\
c_9 &: \{\neg \Delta^- p(X), p(X)\} && \text{(step 3)}\\
c_{10} &: \{\neg \delta^- p(X), p(X)\} && \text{(step 3)}\\
c_{11} &: \{\neg \Delta^+ p(X)\} && \text{(step 4)}\\
c_{12} &: \{\neg \Delta^+ \bot\} && \text{(step 5)}
\end{aligned}$$

We can obtain the empty clause by resolving the clauses in C as follows.

$$\begin{aligned}
d_1 &: \{\neg \Delta^+ p(b)\} && \text{(resolve } c_2, c_{12})\\
d_2 &: \{\} && \text{(resolve } d_1, c_{11})
\end{aligned}$$

Therefore, the supplied update policy satisfies the constraint on the database.

Complexity: We can reduce the problem of deciding equivalence of two Datalog programs to VERIFY-POLICY. We present an outline of such a reduction is as follows. Consider two Datalog queries q_1 and q_2 that are defined using programs P_1 and P_2 respectively. We generate a VERIFY-POLICY instance with a single constraint $\bot$:- b, and the update rule $\Delta^+ a$:- $\delta^+ a$, $\neg b$, and the rules

$\Delta^+r(\bar{X})$:- $\delta^+r(\bar{X})$ and $\Delta^-r(\bar{X})$:- $\delta^-r(\bar{X})$ for every base relation r in $P_1 \cup P_2$, where b is defined using the rules in P_1 and P_2, and the following.

$$b \text{ :- } q_1(\bar{X}),\ \neg q_2(\bar{X}).$$
$$b \text{ :- } q_2(\bar{X}),\ \neg q_1(\bar{X}).$$

In this case, answer to the generated instance of VERIFY-POLICY is *Yes* if and only q_1 and q_2 are equivalent. Since deciding equivalence of two Datalog programs is undecidable [Shm93], we have the following intractability result.

Theorem 2. *In a general setting, VERIFY-POLICY is* undecidable.

Theorem 3. *If the policies and constraints are unions of conjunctive queries, then VERIFY-POLICY is NP-hard. If the policies and constraints contain negation or inequalities i.e.* $\Delta^-, \leq, \geq$*, then VERIFY-POLICY is* Π_2^P*-hard.*

Proof of the theorem follows directly from our reduction, and the complexity results presented in [Ull97; SY80].

5 Conclusion and Outlook

In this paper, we have investigated the problem of expressing policies for updating databases. We gave formal definition of minimal inclusive update policies and presented an expressive declarative language to enable administrators to defining their update policies. While update policies help defining the desired database behavior in ambiguous situations in a powerful and flexible way, an update policy that can lead to an illegal database state is often not desirable. In order to support administrators in authoring update policies that are consistent with the database constraints, we have also presented a resolution-based technique to check update policies for their soundness and completeness wrt. database constraints.

An alternative or complementary approach to the a-posteriori verification is to allow administrators to create update policies from the constraints interactively. Such an update policy authoring process can be facilitated by automatically generating all necessary and sufficient update policies and letting the administrator to pick one policy or combine multiple policies.

In this paper, we have considered only static database constraints. Therefore, every transaction that leads a database to a legal database state was a legal transaction. In general, a database may also have the so-called dynamic constraints which restrict the set of legal transactions. In future, we plan to extend the presented approach to cover dynamic constraints as well.

References

[FN71] Richard Fikes and Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving". In: *Artif. Intell.* 2.3/4 (1971), pp. 189–208.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. "Equivalences Among Relational Expressions with the Union and Difference Operators". In: *J. ACM* 27.4 (1980), pp. 633–655.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989. ISBN: 0-7167-8162-X.

[Shm93] Oded Shmueli. "Equivalence of Datalog queries is undecidable". In: *The Journal of Logic Programming* 15.3 (1993), pp. 231–241.

[WC96] Jennifer Widom and Stefano Ceri, eds. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN: 1558603042.

[Ull97] Jeffrey D. Ullman. "Information Integration Using Logical Views". In: *Proceedings of the 6th International Conference on Database Theory*. Springer-Verlag, 1997, pp. 19–40. URL: `http://dl.acm.org/citation.cfm?id=645502.656100`.

[GLC99] Benjamin N. Grosof, Yannis Labrou, and Hoi Y. Chan. "A declarative approach to business rules in contracts: courteous logic programs in XML". In: *EC*. 1999, pp. 68–77.

[BB01] Craig Boutilier and Ronen I. Brafman. "Partial-Order Planning with Concurrent Interacting Actions". In: *J. Artif. Intell. Res. (JAIR)* 14 (2001), pp. 105–136.

[Orm01] Levent V. Orman. "Transaction Repair for Integrity Enforcement". In: *IEEE Trans. Knowl. Data Eng.* 13.6 (2001), pp. 996–1009.