Update Policies

Abhijeet Mohapatra, Sudhir Agarwal, and Michael Genesereth

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

Abstract. Underspecified transactions can be supported in databases by enabling administrators to specify update policies that complete underspecified transactions. 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 with respect to database constraints 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 with respect to database constraints.

1 Introduction

Many software systems use databases to model the state of the world. A database consist of base relations and derived relations (also called as *views*). Transactions on a database change the state of the database. A database may also contain *constraints* which characterize the set of allowable or *legal* database states and transactions.

For ease of use it is desirable to allow users to submit underspecified transactions. In such scenarios, database constraints may contain sufficient information for completing an incomplete transaction unambiguously. Otherwise, there are multiple ways to complete an underspecified transaction.

Underspecified transactions may either be legal or illegal. Database management systems typically reject illegal transactions. Prior works on repairing illegal transactions propose strategies for generating necessary and sufficient conditions for repairs. A survey of transaction repair strategies is presented in [4]. However, the proposed transaction repair techniques do not incorporate administrators' preferred strategies for completing underspecified transactions. A related approach is that of Courteous Logic Programs [3]. Courteous Logic Programs enable administrators to prioritize how different conflicts are resolved when evaluating queries on a database state. However, strategies for repairing or completing illegal transactions involve two database states.

In this paper, we investigate the problem of supporting underspecified transactions in database management systems. We present a formal framework for expressing *update policies* for updating databases. We identify a class of update policies called *inclusive* update policies which correspond to strategies for completing underspecified transactions. Transactions on databases with constraints and arbitrary update policies may potentially lead to illegal database states.

To prevent such scenarios, we present a resolution-based technique to verify whether or not the specified update policies are sound and complete with respect to database constraints.

2 Problem Definition

A relational 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 D 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 characterize 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, as shown in the following example, 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 which 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*. The relation *on* characterizes whether or not a block is placed directly on another block. The relations *table* and *clear* characterize whether or not a block is on the ground and whether or not a block has other blocks on it respectively.

Suppose blocks a and b are initially placed on the ground.The corresponding database instance contains the atoms $\mathit{table}(a)$, $\mathit{table}(b)$, $\mathit{clear}(a)$, and $\mathit{clear}(b)$.

Consider an action in the blocks world that places block a is on block b. This action corresponds to the insertion of the atom $\mathit{on}(a, b)$ in the database. However, this correspondence does not explicitly specify whether or not the atoms $\mathit{table}(a)$ and $\mathit{clear}(b)$ are deleted from the database. 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. multiple supersets of the input that are legal transactions. For example, suppose a third block c is added to the model of the blocks world in Example 2. Suppose that analogous to blocks a and b, c is 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 deal with scenarios where users specify a subset of the intended transaction, we present a framework for administrators to author strategies for constructing *completed transactions*. 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 $R_d \subseteq D$.*

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 framework, an inclusive policy corresponds to an administrator's strategy for augmenting transaction requests, which may potentially be underspecified, 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 1. *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 database instance D. There are two cases depending on whether or not there exists a pair of sets A and B such that $A \subseteq T_i$, $B \subseteq T_d$, $\langle A, B \rangle$ is a legal transaction on D, and $1 \leq |A| + |B| \leq k$. If there does not exist a pair of sets A and B that satisfy this property, then we construct a minimal policy by assigning $\langle T_i, T_d \rangle$ to some singleton sub-transaction on D. Otherwise, let D_1 be the database instance $(D \setminus B) \cup A$. By definition, D_1 is legal. Therefore, the transaction $\langle T_i \setminus A, T_d \setminus B \rangle$ is legal on D_1, and the theorem can be proved by applying the induction hypothesis on the transactions $\langle A, B \rangle$ and $\langle T_i \setminus A, T_d \setminus B \rangle$ on D and D_1 respectively.

3 Update Policy Language

In this section, we present a language that enables administrators to expressively specify update policies. Our update policy language is Datalog^u, which is an extension of standard Datalog with negation as failure, and *update* operators. First, we present a brief overview of Datalog. Then, we discuss the syntax of Datalog^u and discuss its expressive power.

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

3.1 Overview of Datalog

A Datalog program is a finite set of *facts* and *rules*. Facts are ground atoms. An atom is a structure with the signature $r(\bar{t})$, where p is a relation constant with arity n, and $\bar{t}$ is a finite sequence of n *terms* $t_1, t_2, \ldots, t_n$. Datalog terms are either object constants or variables. Relation constants and object constants are denoted as strings which begin with a lowercase character e.g. *on*, *a*, and variables are denoted as strings which begin with an uppercase character e.g. X, Y. An atom is *ground* if it has no variables.

A Datalog rule is an expression of the form H :- $B_1, B_2, \ldots, B_k$, where k is finite and $k \geq 1$. Such a rule comprises of a distinguished atom H, which is the also called the *head* of the rule, and literals B_1, B_2, ..., and B_k, which comprise the *body* of the rule. A Datalog literal is an atom or the negation of an atom. In what follows, we represent negated atoms as atoms prefixed by the '$\neg$' symbol 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.

Semantically, a rule states that the conclusion of the rule (denoted as the head) is true whenever the conditions (denoted as the literals in the body) are true. Consider the following Datalog rule.

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

Here, $r(X,Y)$ is the head of the rule. The body of the rule consists of the literals $p(X,Y)$ and $\neg q(Y)$. 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 $r(a,b)$ is true.

To ensure finite termination and unique minimal models of Datalog programs *safety* and *stratification* restrictions [7] are imposed on Datalog rules. A safe and stratified Datalog program can be evaluated in a top-down or in a bottom-up fashion [7] in time that is polynomial in the size of the program.

3.2 Constraints in Datalog

Constraints in a Datalog program are encoded 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 Datalog^u: Extending Datalog with Update Operators

We propose to model update policies as Datalog^u programs. Datalog^u extends Datalog (with negation as failure) using four update operators δ^+, δ^-, Δ^+ and Δ^-. Each of these update operators takes an atom as an argument.

Update policies are encoded in Datalog^u as follows. The update operators δ^+ and δ^- are used to respectively encode insertions and deletions in a transaction request R. The operators Δ^+ and Δ^- are used to respectively encode the insertions and deleteinos in a transaction T. 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.

An update policy (see Definition 4) is modeled as a safe stratified Datalog^u program which defines Δ^+ and Δ^- in terms of δ^+, δ^- and the current database instance.

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

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

The above policy ensures that if a transaction request requests the insertion of $p(b)$, then the next database instance contains both $p(a)$ and $p(b)$. This policy also ensures that if a transaction request requests the deletion of $p(a)$, then $p(b)$ is also deleted if $p(b)$ is present. We note that additional rules are not needed to handle transaction requests for insertion of $p(a)$ and deletion of $p(b)$. This is because the resulting database instance is legal.

3.4 Expressiveness of Datalog^u

As discussed in Section 2, 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, Datalog^u, can model all the functions since we can enumerate all the functions.

In case there are countably (infinitely) many database instances, there are also countably infinite transaction requests and countably infinite transactions. In this case, the number of possible update policies is uncountably infinite. Therefore, Datalog^u cannot model all possible update policies.

However, Datalog^u is Turing Complete, i.e., it can model all recursively enumerable functions. Functions which cannot be modeled using our Datalog^u also cannot be modeled by any other programming language.

Comparison with ECA-Rules: Database management systems use an approach that is similar to ours in enforcing constraints through *triggers*. Triggers are event-condition-action (ECA) rules [9] 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. ECA-rules can be modeled as Datalog^u rules by characterizing the triggering event as a transaction request, the triggering condition as a query, and the triggered action as a transaction.

Our update policy language, Datalog^u, is *strictly more expressive* than the language used in triggers. In the following example, we present an update policy which cannot be modeled as a trigger.

Example 5. Consider a database with three unary base relations p, q and r, and the following update policy.

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

The above update policy inserts tuples of the form $q(X)$ in response to transaction requests which request the insertion of tuples of the form $p(X)$, and the deletion of tuples of the form $r(X)$. Such an update policy *cannot* be implemented using triggers without changing the database's schema e.g. by introducing auxiliary tables. This is because triggers are executed in response to a *single* type of event e.g. insert, update on *single table*.

Comparison with STRIPS Action Representation: The STRIPS [2] 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 modeled as an update policy as follows. The update policy contains one rule for each element in the add and delete lists. These rules satisfy the following properties. The head of a rule is $\Delta^{+}e$ or $\Delta^{-}e$ depending on whether an element e is in the add list or the delete list. The body of a rule is a conjunction of the atoms listed as the preconditions of the action, and $\delta^{+}a$ where a is an atom representing the action. In the following example, we model a STRIPS representation of an action in the blocks world as an update policy.

Example 6. Consider the modeling of the blocks world presented in Example 2. Suppose we denote the act of placing a block X on a block Y as $place(X, Y)$. The STRIPS representation of $place(X, Y)$ is as follows.

Action	**Preconditions**	**Add List**	**Delete List**
$place(X, Y)$	$clear(X),\ clear(Y)$	$on(X, Y)$	$clear(Y)$
$place(X, Y)$	$clear(X),\ clear(Y),\ on(X, U)$	$on(X, Y)$, $clear(U)$	$clear(Y)$, $on(X, Y)$

The STRIPS representation of $place(X, Y)$ can be equivalently modeled using the following update policy.

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

The presented update policy ensures the following behavior. First, requests for placing a block X on another block Y are completed only when both blocks are clear i.e. there is no other block on top of the blocks. Second, the block Y is no longer clear in the next state. Third, in the case that block X is on a block, say U, then X is no longer on U in the next state of the blocks world.

Comparison with a STRIPS Extension for Joint-Actions: Prior work presented in [1] presents a formalism for representing joint-actions and generating concurrent non-linear plans by extending STRIPS representations of actions. Joint-actions are represented by adding a possibly empty *concurrent list* to each action. A concurrent list specifies which actions may or may not co-occur with the given action in order to produce the described effect. Such joint-actions can be modeled as transaction requests in Datalog^u since multiple δ^+ and δ^- literals allowed in the body of Datalog^u rules.

In the formalism presented in [1], the concurrency lists and the effect of concurrent execution of actions a and b needs to described twice, once each for actions a and b. The concurrency list of a contains b and the concurrency list of b contains a, and analogously the effects. With Datalog^u we need to define the effect of concurrent execution of a and b only once.

In addition to and despite the simpler syntax of Datalog^u, Datalog^u is more expressive than the formalism presented in [1]. In Datalog^u, we can define effects that are recursively computed (see example below) from the current database instance and the requested transaction. Such effects cannot be modeled with the formalism presented in [1].

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

Another interesting difference between the formalism presented in [1] and our update policy language, Datalog^u, concerns multiple occurrences of the same action. While the former does not allow multiple concurrent actions by the same agent, Datalog^u can deal with multiple occurrences of an action.

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 $\langle T_i, T_d \rangle$ for every transaction request $\langle R_i, R_d \rangle$.*

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* $D \cup \Lambda \not\vdash \bot$. 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 [4], 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 used 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 7. Consider a database that contains the following static constraint $\perp$:- $p(b)$. 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.

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

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

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

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

Complexity: The problem of deciding equivalence of two Datalog programs is reducible to VERIFY-POLICY. We present an outline of such a reduction in what 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 $\perp$:- 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 [6], 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 [8, 5].

5 Conclusion and Outlook

In this paper, we have investigated the problem of supporting underspecified transactions in database management systems. We have presented a formal framework for expressing *update policies* for updating databases. We have identified a class of update policies called *inclusive* update policies which correspond to strategies for completing underspecified transactions. We have also presented a language, called, Datalog^u for expressing update policies.

Transactions on databases with constraints and arbitrary update policies may potentially lead to illegal database states. To prevent such scenarios, we have proposed a resolution-based technique that verifies whether or not specified update policies are sound and complete with respect to database constraints.

An alternative or complementary approach to the a-posteriori verification is to have administrators author update policies from the constraints in an interactive manner. Such an update policy authoring process can be facilitated by automatically generating all necessary and sufficient update policies, and letting the administrator pick one policy or combine multiple policies. We also note that the process of authoring minimal inclusive update policies can be simplified through a rule IDE that automatically adds rules of the form $\Delta^+ r(\bar{X})$:- $\delta^+ r(\bar{X})$ and $\Delta^- r(\bar{X})$:- $\delta^- r(\bar{X})$.

In this paper, we have considered only static database constraints. In such scenarios, every transaction between two legal database states is legal. In general, a database may also have contain *dynamic constraints* which place additional restrictions on legality of transactions. In the future, we plan to extend the presented approach to cover dynamic constraints.

References

1. Boutilier, C., Brafman, R.I.: Partial-order planning with concurrent interacting actions. J. Artif. Intell. Res. (JAIR) 14, 105–136 (2001)
2. Fikes, R., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artif. Intell. 2(3/4), 189–208 (1971)

3. Grosof, B.N., Labrou, Y., Chan, H.Y.: A declarative approach to business rules in contracts: courteous logic programs in XML. In: EC. pp. 68–77 (1999)
4. Orman, L.V.: Transaction repair for integrity enforcement. IEEE Trans. Knowl. Data Eng. 13(6), 996–1009 (2001)
5. Sagiv, Y., Yannakakis, M.: Equivalences among relational expressions with the union and difference operators. J. ACM 27(4), 633–655 (1980)
6. Shmueli, O.: Equivalence of datalog queries is undecidable. The Journal of Logic Programming 15(3), 231–241 (1993)
7. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume II. Computer Science Press (1989)
8. Ullman, J.D.: Information integration using logical views. In: Proceedings of the 6th International Conference on Database Theory. pp. 19–40. Springer-Verlag (1997), http://dl.acm.org/citation.cfm?id=645502.656100
9. Widom, J., Ceri, S. (eds.): Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)