Modal Queries for Relational Databases

C. J. Fox
B. G. T. Lowden
R. Turner

Abstract

This is an exposition of the formal theory of the evaluation of modal queries with respect to relational databases. Such queries allow a database user to effectively interrogate a combination of both the facts and the integrity constraints, associated with the database, and to determine putative extensions to a database with queries expressed in relational query languages (the Tuple Relational Calculus in this case) extended with a modal operator. This work was done as part of a SERC funded research project\(^1\).

Department of Computer Science
University of Essex
Wivenhoe Park
Colchester
CO4 3SQ
U.K.

Tel.no. +44 (0)206 873333
Fax.no. +44 (0)206 873598

\(^1\) Natural Language Front-ends (to Relational and Logic Databases); a Formal Semantics Approach, a three year project supported by SERC grant GR/E/69485 from October 1988 to September 1991. Grant holders: Anne De Roeck, Barry Lowden and Prof. Ray Turner; Senior Research Officers: Bryan Walls, H.E. Jowsey and Chris Fox.
1.0 Introduction

As part of a project on natural language front-ends to databases [4] [5] [14], some initial ideas concerning modal queries in relational databases were explored [12] [13]. In a relational database [3], there are sets called relations which have tuples as members. A tuple is a set of ordered pairs. These ordered pairs consist of attribute-value pairs. The value associated with an attribute is drawn from a domain that is fixed by the attribute.

Standardly, all the tuples in a relation must have the same set of attributes. We shall have to relax this to allow for undefined values. The tuples in a relation can be identified by a key. This is a value (set of values) of the attribute (attributes) which is unique for each tuple. To simplify the following theory, it shall be assumed that relations have only one attribute as a key. Here there may be instances where the key is undefined, in which case it must be possible to refer to tuples directly, without the use of a key. This requires variables and constants that range over tuples; hence Tuple Relational Calculus (TRC) [3].

TRC is can be used to express entries in a relational database, and queries about those relations. Standardly, a query in TRC is evaluated with respect to one database. In this paper however, a the addition of a modal operator to TRC is considered. Its semantics are given in terms of accessible database states (cf. possible worlds [6] [8] [11]). This notion of accessibility is elucidated in terms of updates. The notion of updating a database is made explicit. Rules are given which allow an appropriate update to be constructed from the modal proposition. This is unlike related work on adding temporal notions to relational databases [2] (we can view temporal operators as a variety of modal operator) where all the required relations already exist, and are not constructed to answer a particular query.

Initially the semantics of TRC is given in terms of an informal meta-language. However, in dealing with the semantics of the modal operator, it was found advantageous to make the syntax of the meta-language explicit. This is called the General Relational Calculus (GRC), a multi-sorted language with quantification over all sorts.

2.0 The Syntax of TRC

The tuple relational calculus has the following terms.

\[
\alpha = \kappa \mid t.a
\]

with relations; r tuples; t attributes; a, and \( \kappa \) the constant values

Queries in TRC are

\[
\theta = \{ <t_1.a_1, \ldots, t_n.a_n> | t_1 \in r_1, \ldots, t_n \in r_n > | p \}
\]
where \( p \), the propositions in TRC are given by:

\[
    \begin{align*}
    p &= \alpha = \alpha' \mid \\
    &\alpha < \alpha' \mid \\
    &\overline{p} \mid \\
    &p \land p' \mid \\
    &p \lor p' \mid \\
    &\text{All } t \in r.p \mid \\
    &\text{Exists } t \in r.p
    \end{align*}
\]

with relations \( r \), tuples variables \( t \), attributes \( a \).

Note that to define a general unique key constraint within TRC, equality between tuples would be needed. The version of TRC used here does not have this.

### 3.0 The Semantics of TRC

Although semantics for relational query languages have been examined before [16], it was more convenient to start afresh, because of the new goal of examining a modal extension.

A model for TRC is \(<I, \|\|, \delta, g>\) where \( \delta \) is database state, and \( \|\| \) is an interpretation function for propositions in TRC. \( g \) assigns values to TRC constants. The function \( I \) gives an interpretation of queries in TRC. A notion of fundamental types such as integers and strings etc. is needed. TRC attribute-values must be drawn from some restricted domain dependent upon the attribute in question. These domains, in turn, are subsets of the fundamental types. A full account of this is given in Section 4.0, where the meta-language is formalised.

\[
I(\{<t_1.a_1, \ldots, t_n.a_n>|<t_1 \in r_1, \ldots, t_n \in r_n>|p\}) = A \iff
\]

there does not exist \( v_1 \ldots v_n \) in database \( \delta \) such that

\[
t_1.a_1 = v_1 \ldots t_n.a_n = v_n \text{ where } t_1 \in r_1 \ldots t_n \in r_n \text{ and }
\]

\[
\|p[v_1/(t_1.a_1)] \ldots [v_n/(t_n.a_n)]\|_\emptyset = e
\]

and

\[
A = \{<v_1, \ldots, v_n>| \text{ in database } \delta
\]

\[
t_1.a_1 = v_1 \ldots t_n.a_n = v_n \text{ where } t_1 \in r_1 \ldots t_n \in r_n \text{ and }
\]

\[
\|p[v_1/(t_1.a_1)] \ldots [v_n/(t_n.a_n)]\|_\emptyset = t
\]

\[
I(\{<t_1.a_1, \ldots, t_n.a_n>|<t_1 \in r_1, \ldots, t_n \in r_n>|p\}) = e \iff
\]

there exists \( v_1 \ldots v_n \) in database \( \delta \) such that

\[
t_1.a_1 = v_1 \ldots t_n.a_n = v_n \text{ where } t_1 \in r_1 \ldots t_n \in r_n \text{ and }
\]

\[
\|p[v_1/(t_1.a_1)] \ldots [v_n/(t_n.a_n)]\|_\emptyset = e
\]
The semantics of propositions in TRC are then:

\[ \| \kappa \| \sigma = g(\kappa) \]
\[ \| t \cdot a \| \sigma = v \text{ iff in database } \delta, t \cdot a = v \text{ (where } t \in r \text{ is recorded in } \sigma) \]
\[ \| t \cdot a \| = e \text{ iff } t \in r \text{ recorded in } \sigma \text{ but } a \text{ is not a valid argument of } r \]

\[ \| \alpha = \alpha' \| \sigma = t \text{ iff } \alpha \text{ and } \alpha' \text{ are of the same fundamental type and} \]
\[ \| \alpha \| \sigma \text{ and } \| \alpha' \| \sigma \text{ are equal, and neither equals } e \]
\[ \| \alpha = \alpha' \| \sigma = e \text{ iff } \alpha \text{ and } \alpha' \text{ are not of the same fundamental type, or} \]
\[ \text{ either } \| \alpha \| \sigma \text{ or } \| \alpha' \| \sigma \text{ equals } e \]
\[ \| \alpha = \alpha' \| \sigma = f \text{ otherwise} \]

\[ \| \alpha \triangleleft \alpha' \| \sigma = t \text{ iff } \alpha \text{ and } \alpha' \text{ are of the same fundamental type and} \]
\[ \| \alpha \| \sigma \text{ is less than } \| \alpha' \| \sigma \text{, and neither equals } e \]
\[ \| \alpha \triangleleft \alpha' \| \sigma = e \text{ iff } \alpha \text{ and } \alpha' \text{ are not of the same fundamental type, or} \]
\[ \text{ either } \| \alpha \| \sigma \text{ or } \| \alpha' \| \sigma \text{ equals } e \]
\[ \| \alpha \triangleleft \alpha' \| \sigma = f \text{ otherwise} \]

\[ \| \bar{p} \| \sigma = t \text{ iff } \| p \| \sigma = f \]
\[ \| \bar{p} \| \sigma = e \text{ iff } \| p \| \sigma = e \]
\[ \| \bar{p} \| \sigma = f \text{ iff } \| p \| \sigma = t \]

\[ \| p \land p' \| \sigma = t \text{ iff } (\| p \| \sigma = t) \text{ and } (\| p' \| \sigma = t) \]
\[ \| p \land p' \| \sigma = e \text{ iff } (\| p \| \sigma = e) \text{ or } (\| p' \| \sigma = e) \]
\[ \| p \land p' \| \sigma = f \text{ otherwise} \]

\[ \| p \lor p' \| \sigma = t \text{ iff neither } \| p \| \sigma \text{ nor } \| p' \| \sigma \text{ equals } e \]
\[ \text{ and at least one equals } t \]
\[ \| p \lor p' \| \sigma = e \text{ iff either } \| p \| \sigma \text{ or } \| p' \| \sigma \text{ equals } e \]
\[ \| p \lor p' \| \sigma = f \text{ otherwise} \]

\[ \| \text{All } t \in r \cdot p \| \sigma = t \text{ iff for all } t' \in r \text{ in } \delta, \| p \sub{t' \rightarrow t} \| \sigma' = t \]
\[ \text{ where } \sigma' \text{ records all in } \sigma \text{ and that } t' \in r \]
\[ \| \text{All } t \in r \cdot p \| \sigma = e \text{ iff there exists } t' \in r \text{ in } \delta \text{ such that} \]
\[ \| p \sub{t' \rightarrow t} \| \sigma' = e \text{ where } \sigma' \text{ records all in } \sigma \text{ and that } t' \in r \]
\[ \| \text{All } t \in r \cdot p \| \sigma = f \text{ otherwise} \]
\[ \exists t \in r.p \mid \sigma = t \text{ iff for some } t' \in r \text{ in } \delta, \| p[ t'/t ] \| \sigma' = t \]
where \( \sigma' \) records all in \( \sigma \) and that \( t' \in r \)
\[ \exists t \in r.p \mid \sigma = e \text{ iff there exists } t' \in r \text{ in } \delta \text{ such that} \]
\[ \| p[ t'/t ] \| \sigma' = e \text{ where } \sigma' \text{ records all in } \sigma \text{ and that } t' \in r \]
\[ \exists t \in r.p \mid \sigma = f \text{ otherwise} \]

The notion of “error” here is strong. If any sub-part of an expression is in error, or any part of the evaluation of a query is in error, then the whole expression is in error. It is possible to weaken this, so that, for example, a disjunction can be true if one disjunct is true, even if the other disjunct is in error (cf. Kleene’s strong 3-valued connectives [9]).

This completes the semantics of “standard” TRC. The addition of a modal operator complicates this considerably.

The method of evaluating a modally interpreted proposition, presented here, essentially tries to find a consistent database state, accessible from the current state, in which the proposition is true [8]. A database state is accessible from the current database state if there is an update that can be applied to the current database state, which results in the required state. A consistent database state is one where all the usual database integrity constraints hold, and no optional constraints are violated.

Further more, it is desirable that all database states considered for accessibility belong to the same class of databases. For example, when evaluating a modal proposition with respect to a database about students and courses, we do not want to consider accessible database states concerned with patients and hospitals. A class of database can be described by a data model.

Ideally, a language is needed in which to express data models. These require quantification over attributes, relations, and databases. TRC, by definition, can only quantify over tuples, so is unsuitable for this purpose. Additionally, TRC is unsuitable for representing all integrity constraints, for example, the unique key constraint [3], cannot be expressed in its general form in TRC.

A language can be defined which allows quantification over all sorts. This can be called the General Relational Calculus (GRC). It turns out that this language can also be used as the meta-language for the semantics of TRC (Appendix B).

### 4.0 The Syntax of GRC

GRC has the terms; 
- domains \( d \); database states \( \delta \); constraints \( \chi \); 
- relations \( r \); tuples \( t \); attributes \( a \); values \( v \)
There are also fundamental types $\tau_1, \ldots, \tau_n$, such as strings, numbers, etc. We can write:

$$T_i(v)$$

when $v \in \tau_i$.

Basic expressions of GRC are given by

$$\beta = [r, t, a, v]_{\delta} \mid [r, t, a, *]_{\delta} \mid t = s \mid t < s \mid A(r, a) \mid D(a, d) \mid K(r, a) \mid T_i(v) \mid \{i \text{ an integer}\} \mid C_\chi(\delta) \mid \delta R \delta'$$

Where $r, t, a, v$ are variables of the appropriate sort. Constants, if they are needed, will be subscripted by $c$. Constants of sort $v$ may be written $\kappa$. The predicates have the following intuitive meanings:

- $[r, t, a, v]_{\delta}$ - the value of attribute $a$ of tuple $t$ in relation $r$ is $v$ in database $\delta$.
- $t = s$ - equality.
- $A(r, a)$ - relation $r$ has attribute $a$.
- $D(a, d)$ - the value of attribute $a$ must come from domain $d$.
- $K(r, a)$ - relation $r$ has attribute $a$ as a key field.
- $T_i(v)$ - value $v$ is of fundamental type $\tau_i$.
- $C_\chi(\delta)$ - database $\delta$ is consistent with integrity constraints $\chi$.
- $\delta R \delta'$ - database $\delta'$ is related to (accessible from) $\delta$.

Partial equality can be defined as follows:

$$t = s =_{def} \neg(t = *) \land \neg(s = *) \land t = s$$

And a term is ground

$$i =_{def} \neg(t = *)$$
The wff of GRC are

\[ \text{wff}; \varphi = \beta \downarrow \]
\[ \neg \varphi \downarrow \]
\[ \varphi \lor \varphi' \downarrow \]
\[ \varphi \Rightarrow \varphi' \downarrow \]
\[ \forall x \varphi \downarrow \]
\[ \exists x \varphi \]

Where \( x \) ranges over one of \( r, t, a, v, d, \delta \). These have the standard first-order logic proof theory.

### 4.1 Database Domains

A database domain can be defined by a set of the predicates \( A, D, K \). Any putative definition of a domain in GRC must be equivalent to some set \( \Gamma \) of ground atomic sentences consisting of predicates \( A, D, K \).

The theory is simplified by assuming that there is only one key for each relation. A database \( \delta \) is a set of

\[ [r, t, a, v]_\delta \]

where \( r \) is from the set of relations mentioned in some \( K \) predicate, \( a \) is an attribute of that relation, as declared by some \( A \) predicate, \( t \) is any valid name, and \( v \) is a value from the domain given for that attribute in that relation by \( D \).

In addition, \( C \) can constrain this set of possible databases further using a set of integrity constraints \( \chi \).

Later, terms of the form

\[ [r, t, a, *]_\delta \]

shall be considered to be legitimate members of a database. * is the null value, it belongs to all domains and all fundamental types. In constraint checking, each * introduced this way behaves like a unique variable. They are effectively instantiated during consistency checking:

\[ C_\chi(\delta) \text{ iff } \forall rta ( [r, t, a, *]_\delta \Rightarrow \exists v (C_\chi(\delta [ [r, t, a, v] / [r, t, a, *])] )) \]

If a set of GRC sentences \( \Gamma \) satisfy the appropriate restrictions for a data model, then \( \text{Dom}(\Gamma) \) holds. The set of databases that are described by \( \Gamma \) is written \( \Delta_\Gamma \). The subset of this where a set of integrity constraints \( \chi \) are satisfied, is written \( \Delta_{\Gamma, \chi} \).

The axioms which govern a data model in GRC are given in Appendix A.
4.2 The Semantics of TRC in terms of GRC

A model for TRC is a domain definition (data model) and an assignment function $\| \cdot \|_\delta$, relativised to a database state $\delta$ for assigning truth values to propositions in TRC. There is a separate interpretation function $I$ for interpreting TRC queries. A model for TRC is thus

$$<I_\delta, \| \cdot \|_\delta, \Gamma>$$

where $Dom(\Gamma)$ and $\delta \in \Delta_\Gamma$

The semantics (without the modal operator) are given in full Appendix B. They are essentially the same as those given informally in 3.0.

5.0 Modality

To add modality to this, a possibility operator must be added to the wff of TRC:

$$\Box \varphi$$

$$\| \Box \varphi \|_\delta \sigma = t \text{ iff } \exists \delta' (\| \varphi \|_\delta = t) \land \delta \mathcal{R} \delta'$$

$$\| \Box \varphi \|_\delta \sigma = e \text{ iff } \| \varphi \|_\delta \sigma = e$$

$$\Box \varphi \|_\delta \sigma = f \text{ otherwise}$$

The major task of this paper is to give an effective definition of the right-hand side of this expression, that is, to be able to effectively produce $\delta'$ from $\delta$ such that we would intuitively agree that $\delta \mathcal{R} \delta'$. A modal proposition is verified by updating the database with information needed to make the proposition true. The modal proposition is true if this new database state is consistent. Helpful answers can be given by stating the 'space' of database states where the proposition can consistently be held.

The notion of accessibility will be defined in terms of updates: possibly $p$ is to be interpreted as is it possible to update the current database to achieve a consistent state in which the proposition $p$ holds?

For the moment, it shall be assumed that the modal operator can only out-scope a TRC proposition: the modal operator does not occur before TRC queries. This restriction means that all free variables in a modal expression will be bound outside that expression, and the semantics of TRC can be restricted so that free variables can only adopt values which occur in the current database state. Later, the relaxation of this restriction will be examined.

5.1 Accessibility as Update

To provide an effective evaluation, it is necessary to state how the appropriate update can be derived from the contents of the proposition. Such an update, when applied to the current state, must make the proposition true. The modal proposition is true if this is a consistent database state. There may be several possible updates, but only one needs to be found which verifies the modal proposition (when dealing with the modality of possibility).
The modal proposition may not give sufficient information to enable an update to be derived that maintains the requirements of the relational model. This is because the proposition may require that some attribute values be restricted in their value, but not force them to be a particular value, or, attributes may not be referred to in the proposition, though they may be required by the relations that are mentioned.

This could be dealt with by generating a set of updates which, together, consider all the possible values of attribute values which are either ambiguous in value, or else required by the data-model, but not referred to in the modal proposition. This means that the sets of accessible databases required by the modal proposition are considered explicitly. Alternatively, the update could leave the database in some ambiguous state which subsumes a set of ground, or full instantiated, states. With finite domains the choice is not clear. With potentially infinite domains such as integers, however, the choice is clear: for effective evaluation it is not possible to consider an infinite number of states. This requires that the update contains null values, which may be restricted by constraints associated with the update. The null value indicates that an attribute of a particular tuple has a value, yet that value is unknown. The new constraints must be derived from the modal proposition.

If it can be shown that a subsuming database is consistent, then the set of subsumed databases is not empty (and, thus, that the modal proposition holds).

Negation presents a potential problem. If there is real, atomic negation within the scope of the modal, then there are two approaches. The update could simply force the removal of the attribute values which are inconsistent with the negated expression. However, this causes problems when trying to ensure that the new state is a valid database: all attributes of a tuple must have a value. It must be ensured that it is not possible to consider the offending value. This requirement leads to the approach adopted: null values are inserted in place of the offending values, and constraints are added which ensure that the undesired values can not be considered. An equivalence is thus created: “Mary can not eat cheese”, becomes equivalent to “Mary can eat something, but it is not cheese’. The update used would be “Mary eats something” where “something” is the null value, and a new constraint “that something is not cheese” would be added.

5.2 Principals for Applying Updates

When applying an update to the current database state to reach a new state, the principle will be adopted here which ensures that the maximum amount of information is preserved from the current state, as long as it is does not contradict the update. When that update contains tuples which do not occur in the current state, then the update can just be added. In the relational databases being considered. If the keyfields of “two” tuples are identical, then all the other attribute values in those tuples must be equal. Obviously, performing an update is complicated when there is a “clash” of keyfield values.

If a generated update has a keyfield value which occurs in the current database, then any attribute values of that tuple, not mentioned in the update, are inherited from the current database. This ensures that new state will be as like the current state as possible. This in-
inheritance can be performed whilst applying the update to the current state, and/or by modifying the update to contain the inherited information.

If a keyfield, in an update, has a null value, then if that null value "subsumes" existing keyfields, then inheritance, as described above, should occur. This causes problems because several different keyfield values will be subsumed, in general, and thus different values must be inherited in each of these cases. In this case, the update can be mapped to a set of updates, which consider existing keyfield values. An update with the null keyfield value will still be a member of the set, but this will just cover the case where there is a new, unique, keyfield value.

6.0 Producing an Update

Here, a formalisation of the production of updates from a proposition is given. First, some simple axioms are given which describe the behaviour of update production with respect to negation of atomic terms.

When dealing with negation of an atomic term (i.e. when the consequence of a proposition is that an attribute of a tuple can not have a particular value) an attribute value is added to the update, and a constraint is also added which prevents it having that value.

The consequences of this are that the following equivalence holds:

\[(\| \phi \|_s \sigma = t) \supseteq \neg \exists r t a v. ([r, t, a, v]_\delta \land \lnot v R c) \iff \]

\[(\| \phi \|_s \sigma = t) \supseteq \exists r t a v. ([r, t, a, v]_\delta \land \lnot v R c)\]

and with constants

\[(\| \phi \|_s \sigma = t) \supseteq \neg \exists r t a. ([r, t, a, \kappa]_\delta) \iff \]

\[(\| \phi \|_s \sigma = t) \supseteq \exists r t a v. ([r, t, a, v]_\delta \land \lnot v = \kappa)\]

To give the intuition behind this; with the proposition "Mary does not eat cheese" we are asserting the negation of cheese eating with respect to Mary. We are not asserting the negation of eating. Further, it can be taken to say that Mary does eat something, but it is not cheese. With a relational database, if we insist on the completeness of tuples, then it would break the conditions in the data model to assert that there is a person who does not eat. For this proposition to mean anything with respect to a database, then the data model would assert that if there is someone, they must eat something².

6.1 Axioms for Producing Updates

Here are axioms that are desirable for forming the database update \( U \), and constraint update \( \chi \) for a proposition.

---

². Obviously, here only single tuples are being considered, so that in this case, people, and what they eat, appear in the same tuple.
The two have to be considered in conjunction to ensure that variables are bound correctly between entries in the database and the constraints.

for all \( rtav. ([r, t, a, v]_U) \) and \( (\forall \delta. [r, t, a, v]_\delta \Rightarrow vRc) \in \chi \) iff

\[ (\| \varphi \|_U\sigma = t) \supset [r, t, a, v]_U \land vRc \]

for all \( rtav. ([r, t, a, v]_U) \) and \( (\forall \delta. [r, t, a, v]_\delta \Rightarrow \neg vRc) \in \chi \) iff

\[ (\| \varphi \|_U\sigma = t) \supset [r, t, a, v]_U \land \neg vRc \]

Now, recursive rules for forming the updates in a manner that conforms with these axioms shall be given.

6.2 Recursive Rules for Forming Updates

Now, recursive rules for forming the updates in a manner that conforms with these axioms are given.

6.2.1 Forming the Database Update

Note that in the case of universal quantification there is a problem with the range of the quantifier. If the update is relativised to database \( \delta \) then only tuples in that database will be considered. Because of the manner in which disjunction is handled, there will in general be several functions\(^3\) \( U_\delta \). It is assumed that all variables have been renamed to avoid clashes.

Note that only dealing with propositions (derived from propositions) expressed in the TRC, it is easiest to give these rules directly in terms of TRC propositions. To give them in terms of the GRC translation of a TRC expression requires care to ensure that the GRC proposition is always unpacked into GRC propositions that are translations of TRC propositions:

Unpacking a TRC proposition directly, a sortal context \( \sigma \) which keeps track of the relations that tuple variables range over is needed. If the update is produced by function \( U_\delta \), then it should satisfy the following axioms:

---

3. The subscript may be dropped on occasions. The context should make it clear whether \( U \) refers to a function whose value is an update, or an update itself.
The ground cases:

\[ U(t.a = \kappa) \sigma = \{ [r, t, a, \kappa] \} \]
where \((t \in r) \in \sigma\)

\[ U(t.a = t'.a') \sigma = \{ [r, t, a, *], [r', t', a', *] \} \]
where \((t \in r) \in \sigma\) and \((t' \in r') \in \sigma\)

\[ U(t.a < \kappa) \sigma = \{ [r, t, a, *] \} \]
where \((t \in r) \in \sigma\)

\[ U(t.a < t'.a') \sigma = \{ [r, t, a, *], [r', t', a', *] \} \]
where \((t \in r) \in \sigma\) and \((t' \in r') \in \sigma\)

Driving in negation:

\[ U(t.a = \kappa) \sigma = U(t.a < \kappa \lor \kappa < t.a) \sigma \]
\[ U(t.a = t'.a') \sigma = U(t.a < t'.a' \lor t.a < t.a) \sigma \]
\[ U(t.a < \kappa) \sigma = U(\kappa < t.a) \sigma \]
\[ U(t.a < t'.a') \sigma = U(t'.a' < t.a) \sigma \]

\[ U(\forall t \in r.p) \sigma = U(\exists t \in r.\bar{p}) \sigma \]
\[ U(\exists t \in r.p) \sigma = U(\forall t \in r.\bar{p}) \sigma \]
\[ U(p \land \bar{p}') \sigma = U(\bar{p} \lor \bar{p}') \sigma \]
\[ U(p \lor \bar{p}') \sigma = U(\bar{p} \land \bar{p}') \sigma \]

Decomposing the proposition:

\[ U(\Box p) \sigma = \emptyset \]
\[ U(p \land \bar{p}') \sigma = U(p) \sigma \cup U(\bar{p}') \sigma \]
\[ U(p \lor \bar{p}') \sigma = U(p) \sigma \lor U(p \lor \bar{p}') \sigma = U(\bar{p}') \sigma \]
\[ U(\exists t \in r.p) \sigma = U(p[t'/t]) (\sigma \cup \{ t' \in r \}) \text{ for some } t' \]
\[ U(\forall t \in r.p) \sigma = \bigcup_{t'; \exists t \in r} U(p[t'/t]) (\sigma \cup \{ t' \in r \}) \]

6.2.2 Forming the Constraint Update

Because of the bindings of variables between the database update, and the additional constraints, it is necessary to make the additional constraints dependent upon the particular update \(U\) given by the function \(U_S\). The function that produces the constraints update\(^4\) required for the database update generated by a function \(U\) shall be \(\chi_U\), and has the following axioms:

4. Footnote 3 applies to \(\chi\).
The ground cases:

\[ \chi(t, a = \kappa) \sigma = [r, t, a, v]_\cup \wedge v = \kappa \]
\[ \chi(t, a = t'.a') \sigma = [r, t, a, v]_\cup \wedge [r', t', a', v']_\cup \wedge v = v' \]
\[ \text{where } (t \in r) \in \sigma \text{ and } (t' \in r') \in \sigma \]
\[ \chi(t, a < \kappa) \sigma = [r, t, a, v]_\cup \wedge v < \kappa \]
\[ \text{where } (t \in r) \in \sigma \]
\[ \chi(t, a < t'.a') \sigma = [r, t, a, v]_\cup \wedge [r', t', a', v']_\cup \wedge v < v' \]
\[ \text{where } (t \in r) \in \sigma \text{ and } (t' \in r') \in \sigma \]

Driving in negation:

\[ \chi(t, a = \kappa) \sigma = [r, t, a, v]_\cup \wedge v \neq \kappa \]
\[ \text{where } (t \in r) \in \sigma \]
\[ \chi(t, a = t'.a') \sigma = [r, t, a, v]_\cup \wedge [r', t', a', v']_\cup \wedge v \neq v' \]
\[ \text{where } (t \in r) \in \sigma \text{ and } (t' \in r') \in \sigma \]
\[ \chi(t, a < \kappa) \sigma = \chi(\kappa < t.a) \sigma \]
\[ \chi(t, a < t'.a') \sigma = \chi(t'.a' < t.a) \sigma \]

\[ \chi(\forall t \in r.p) \sigma = \chi(\exists t \in r, \bar{p}) \sigma \]
\[ \chi(\exists t \in r.p) \sigma = \chi(\forall t \in r, \bar{p}) \sigma \]
\[ \chi(p \Lambda p') \sigma = \chi(\bar{p} \lor \bar{p}') \sigma \]
\[ \chi(p \lor p') \sigma = \chi(\bar{p} \Lambda p') \sigma \]

Decomposing the expression:

\[ \chi(\square p) \sigma = \emptyset \]
\[ \chi(p \Lambda p') \sigma = \chi(p) \sigma \land \chi(p') \sigma \]
\[ \chi(p \lor p') \sigma = \chi(p) \sigma \iff U(p \lor p') \sigma = U(p) \sigma \]
\[ \chi(p \lor p') \sigma = \chi(p) \sigma \text{ otherwise} \]
\[ \chi(\exists t \in r.p) \sigma = \chi(p[t'/t]) \sigma \cup \{t \in r\} \text{ iff } \]
\[ U(\exists t \in r.p) \sigma = U(p[t'/t]) \sigma \cup \{t' \in r\} \]
\[ \chi(\forall t \in r.p) \sigma = \forall t' (\exists a' (\{r, t', a, v\} \cup \sigma \cup \{t' \in r\}) \Rightarrow \chi(p[t'/t]) \sigma \cup \{t' \in r\}) \]

Note that when adding these constraints to a database \( \delta \), a global substitution of \( [\delta/U] \) must be performed.

### 7.0 Null Keyfield Values

The are several functions \( U \), depending upon the choices made with disjuncts and existential quantification, and hence, several corresponding updates to be tried.
To make each tuple in the updates complete, null attribute values must be inserted where a particular attribute was not mentioned in the proposition. However, as mentioned before, there are complications with regard to null keyfield values. When applying and update to a database, it is to be assumed that as much information from the original database state will be preserved as possible. This requires the inheritance of values from tuples whose keyfields clash with keyfields in the update. The inheritance is performed when the update is applied. It only occurs if a keyfield explicitly clashes.

With null valued keyfields in the update, there is a potential clash with all the current keyfields in a relation. To cope with this situation, all the possible inheritances that might occur must be considered explicitly. So for each possible update, a set of updates must be generated in which all possible keyfield clashes are considered explicitly. This results in a set of sets of possible updates.

This can be viewed as follows; a modal proposition under-specifies a possible update that can make that proposition true. Because a domain can be infinite, it is best to leave attribute values as unspecified if possible. This is not possible if the attribute is a key-field. When a key-field is unspecified, we must consider it either to have an existing value, or a new value. The unique key constraint comes into effect if we consider an existing value.

*Complete* takes an update and produces a set of updates with null attribute values inserted to make each tuple complete, and all the various instantiations of null keyfields are produced, and inheritance is enforced.

The definition required is as follows: all grounded attribute-values in $U$ appear in $S$. If a null value appears in $U$ and it is not a keyfield, then it appears in $S$ (as a null value). If it is a null keyfield, then it either appears in $S$ (as a null value), or it is replaced with an existing (grounded) keyfield value occurring in $\delta$. If a tuple in a relation appears in an update, without there being a value for one of its required attributes, then either a null value is inserted for that attribute, or if it is a keyfield, then an existing (grounded) keyfield value may be inserted. Whenever an existing keyfield value is used, then all the other attributes in that tuple, which are not mentioned in the original update, must inherit values from the

---

5. Here it is assumed that unmentioned keyfield attributes of a tuple mentioned in the update can be considered to have the values of existing keyfields, in which case inheritance must occur with any other unmentioned values in the same tuple.
tuple with that keyfield in the current database. The final clause ensures that attributes in a
tuple have unique values.

\[
Complete_\delta(U) = \\
\{ S \} \\
\forall rtav([r, t, a, v]_{U} \land \psi \Rightarrow [r, t, a, v]_{S}) \\
\land \\
\forall rta([r, t, a, \ast]_{U} \land \neg K(r, a) \Rightarrow [r, t, a, \ast]_{S}) \\
\land \\
\forall rta([r, t, a, \ast]_{U} \land K(r, a) \Rightarrow \\
([r, t, a, \ast]_{S} \land \exists t'v'([r, t', a, v']_{S} \land [r, t, a, v']_{S})) \\
\land \\
\forall rtaa'v([r, t, a', v]_{U} \land A(r, a) \land \neg \exists v'.([r, t, a, v']_{U}) \Rightarrow \\
[r, t, a, \ast]_{S} \land \\
\exists t''v''(K(r, a) \land [r, t', a, v'']_{S} \land [r, t, a, v'']_{S}) \\
\land \\
\forall rtav\exists t'([r, t, a, v]_{S} \land K(r, a) \land [r, t', a, v]_{S} \Rightarrow \\
\forall a'(-\exists v''( [r, t', a', v'']_{U} \Rightarrow \\
\exists v'( [r, t', a', v']_{S} \land [r, t, a', v']_{S})) \\
\land \\
\forall vv'([r, t, a, v]_{S} \land [r, t, a, v']_{S} \Rightarrow v = v' )
\]

8.0 Applying Updates

For a modal proposition \( \square \phi \) to be true in database state \( \delta \), the proposition itself must be
true in \( \delta \) updated with a member of \( Complete_\delta(U_\delta(\phi)) \), where \( U_\delta \) belongs to the family
of functions described above, to yield a new database state \( \delta' \). In addition, the constraints
associated with \( \delta \) must be updated with \( \chi_{U_\delta}(\phi) \), the constraints associated with \( U_\delta \).
Where these new constraints explicitly refer to the update, they must be made to refer to the
new database state.

Adding the new constraints is straight-forward, if the constraints associated with \( \delta \) is \( \chi \),
then the constraints associated with \( \delta' \) will be

\[
\chi \cup \chi_{U_\delta}(\phi)[\delta'/U_\delta].
\]

However, when considering nested modals, it may be useful to drop the additional con-
straints associated with \( \delta' \) when considering states \( \delta'' \) accessible from \( \delta' \).
\textit{Clash} enforces the general constraint all databases that each attribute in a tuple may only have one value:

\[
\text{Clash}_\delta(U) = \text{def} \quad \{ [r, t, a, v] \mid \\
\exists t' a' v' v'' . ([r, t', a', v']_\delta \land [r, t, a', v']_\delta \land K(r, a) \land \\
[r, t, a, v]_\delta \land [r, t', a, v'']_U \}
\]

\[
\text{Update}_U(\delta) = \text{def} \ (\delta \setminus \text{Clash}_\delta(U')) \cup U' \text{ where } U' \in \text{Complete}_\delta(U)
\]

The updated database \( \delta' = \text{Update}_U(\delta) \). These definitions ensure that all information in the original database is inherited by the new database, as long as there is no explicit contradiction in the update. The main complications arise when an existing key is mentioned in the update.

We now have the following lemma:

\[
\forall rta \delta . ([r, t, a, v]_U \land \delta = \text{Update}_U(\delta) \Rightarrow [r, t, a, v]_\delta)
\]

We can also define \( \delta \mathcal{R} \delta' \) as

\[
\exists U . (\text{Update}_U(\delta) = \delta')
\]

The modal can be evaluated as follows:

\[
\ll [\Box \Phi]_\delta = t \text{ iff } \ll \Phi)_\delta = t \\
\text{where } \delta' = \text{Update}_U(\delta) \\
\text{where } U \in \text{Complete}_\delta(U_\delta(\phi)) \\
\text{where } U_\delta \text{ is some update function}
\]

If the constraints for \( \delta \) are \( \chi \), then the constraints for the database state \( \delta' \) are

\[
\chi \cup \chi_U(\phi) [\delta'/U']
\]

\textbf{9.0 The Strength of the Logic}

The logic described so far corresponds to S4, update can add new tuples but, they can not reduce the number of tuples. This means that the accessibility relation between database states is not symmetric (which is required for S5). With inequalities, however, it may be possible to simulate the deletion of tuples by stating that a tuple may not have the values of the tuple to be "deleted". Whether this really leads to an S5 logic needs further work.

The Barcan formula does not hold. Null values may subsume values which do not occur in the current database, and further, if the null value is in a keyfield position, then it cannot adopt existing values without violating the unique key constraint.
10.0 Modals Outscoping Free Variables

One issue not dealt with is the case where the modal operator has wide-scope over an entire query (as opposed to a proposition). A query is a set definition in TRC. The meaning of a query is the extension of that set definition. This is effective for non-modal queries, and queries where the modal does not out-scope a TRC query, for example

\[ \{ t . a | t \in r \models \square \phi \} , \]

because a database is finite, and extensions of sets can only be finite. If the modal operator out-scopes a query, i.e. it out-scopes the abstracted free variables:

\[ \square \{ t . a | t \in r \models \phi \} , \]

then an answer may be the possible sets of tuples which satisfy the query. This is an incorrect formalisation of such a query, because the modal operator can only prefix propositions, not sets. It should be written

\[ \{ s | \square (s = \{ t . a | t \in r \models \phi \}) \} \]

The syntax of TRC would have to be extended to allow such queries.

However if the queried domains are infinite, then the extensions of these sets can be infinite, and the number of such answers may be infinite. An alternative is to just see if the proposition in the modal set definition is consistent with the constraints, i.e. to check whether the modal set can exist according to the constraints.

A notion of helpfulness may be of use here: rather than just say that there is some possible set of tuples which may satisfy the proposition, it could be more useful to show how the constraints interact with the proposition in the query, to show the further restrictions to possible tuples, needed to ensure that the constraints are satisfied. A notion of helpfulness may additionally be useful for all modal queries

11.0 Helpful Answers

In this setting it can be assumed that a helpful answer is some minimal set of information that enables the actual set of ground databases that satisfy the proposition to be inferred, or in the case of a negative answer, helpful information would indicate which constraints had been violated [1] [17].

The various kinds of helpfulness that may be of use are as follows
1. Indicating further restrictions on tuples when a modal outscopes free variables.
2. Indicate further restrictions on tuples for "ordinary" modal queries.
3. If the answer to a modal is in the negative, to give the constraints which cause the failure to verify the query.
Both (1.) and (2.) may be solved in similar fashion: the proposition in the query $\varphi$ can be strengthened to $\varphi'$, so that is a tuple satisfies $\varphi'$, iff that tuple satisfies $\varphi$ and the constraints associated with the database $\chi$. $\varphi'$ is returned as the "helpful" part of the answer.

Given the query

$$\{ s | s = \{ t.a | t \in r | t.a > 0 \} \}$$

and constraint

$$\forall t \delta ( [r, t, a, v] \delta \Rightarrow v > 10 \land v < 100)$$

we would like the answer

$$\{ t.a | t \in r | t.a > 10 \land t.a < 100 \}$$

or

$$v \in Answer \iff v > 10 \land v < 100$$

This notion of helpfulness can also be extended to the more straight-forward queries so that constraints on the possible values of unspecified attribute values are made explicit in the reply, perhaps with the use of a biconditional as above. Given the query (over the same relation as above)

$$\{ <> | <> | \exists t \in r.t.a' = \kappa \}$$

with just the constraint given above, the answer might be

$$<> \in Answer \iff t.a > 10 \land t.a < 100$$

For (3.) it is possible to trace the original constraints that led to a contradiction with $\varphi$.

### 12.0 Relationship to Deductive Databases

This work is different from existing work on theorem proving applied to databases [7]. Typically in this work, a database is considered to consist of an Extensional Data Base (EDB) and an Intensional Data Base (IDB), which may, or may not, be considered as a separate object. The EDB consists of ground, explicit relations, and the IDB consists of general rules about those facts which enable new information about implicit relations to be derived.

There is a connection between the IDB and integrity constraints, in that integrity constraints are also general rules about the EDB. However, there is no (non-modal) sense in which integrity constraints can be used, in answer to a query, to deduce new information not explicit in the EDB (assuming that statements involving basic relations such as $<$ and $=$ are not considered to give new information). Integrity constraints are syntactically like deductive rules used in theorem proving applied to databases, yet no new relational information can be deduced about the current database state from them [15].
Kowalski says that integrity constraints do give more information about the state of the database because they enable the truth of a special query, \textit{inconsistent}, to be checked [10]. However, such a query cannot be expressed in TRC.

Even if general deductive rules (as opposed to integrity constraints) were attached to the database, the semantics for the non-modal TRC, given here, could not make use of these rules, and additionally, the semantics envisioned for the modal TRC will not be able to use such deductive rules either.

13.0 Summary

This work gives a formal underpinning to the notion of modality as applied to relation databases. Hopefully, the actual implementation of a modal query language based on this theory should be straight-forward (excluding problems of efficiency).

14.0 Future Work

The formal notion of helpfulness sketched here requires further work.

There are other kinds of queries which it might be possible to deal with using extensions to this approach to modal queries, namely \textit{hypothetical} and \textit{counterfactual} queries. These are conditional questions of the form; "if $A$ were the case, would $B$ hold?". In terms of a database, we may view counterfactuals as having antecedents which contradict the database, and hypotheticals as those where the antecedent contradicts the constraints and data model. The implication in such conditional questions is not material implication, otherwise the query would often be trivially true. Rather, the conditional is to be seen as a complex modal, the consequent must be evaluated in a state where the antecedent holds. In the case of counterfactuals, some entries in the database would be overturned, before the consequent was evaluated, and with hypotheticals some of the constraints would have to be overturned.
Appendix A  Axioms for Data Models in GRC

Here are the axioms that govern $A, D, K, C$ when they occur in the definition $\Gamma$ of a database domain

\[
\forall a d'. (D(a, d) \land D(a, d')) \Rightarrow D(a, d \cup d')
\]
\[
\forall a d.(D(a, d) \Rightarrow \exists i. \forall v d. T_i(v))
\]
\[
\forall r a a'. ((K(r, a) \land K(r, a'))) \Rightarrow a = a'
\]
\[
\forall r a. (A(r, a) \Rightarrow \exists a'. K(r, a'))
\]
\[
\forall r a. (A(r, a) \Rightarrow \exists d. D(a, d))
\]

A more sophisticated analysis would allow complex keys to be formed from several attributes in a relation.

We must prevent a putative domain definition from effectively containing disjuncts; we should be prevented from saying

\[
A(r, a) \lor A(r, a')
\]

when neither

\[
A(r, a) \quad \text{nor} \quad A(r, a')
\]

hold.

This can be accomplished with axioms of the following form:

\[
(\Gamma \models \phi \lor \varphi) \text{ implies } (\Gamma \models \phi) \quad \text{or} \quad (\Gamma \models \varphi)
\]

Where $\Gamma \models \varphi$ means that $\varphi$ is a consequence of the set of sentences $\Gamma$.

If a set of GRC sentences $\Gamma$ has all of these properties, then it constitutes a data model, or database domain definition. This can be written:

\[
Dom(\Gamma)
\]

A database $\delta$ is a member of the domain $\Delta$, specified by a domain definition $\Gamma$ iff;

for all $rtav. [r, t, a, v]_\delta$ implies $(\Gamma \models \exists aA(r, a))$

for all $rtav. [r, t, a, v]_\delta$ implies $(\Gamma \models \exists dD(a, d) \land v \in d)$

Additionally, there are axioms that concern databases in general, namely, the uniqueness of attribute values:

\[
\forall r t t' a v v'. ([r, t, a, v]_\delta \land [r, t', a, v']_\delta \Rightarrow t = t' \land v = v')
\]

Standardly, all the attributes specified for a relation must have a value in each tuple. For this exposition it will be convenient to drop this requirement at times. If a database satisfies this requirement, it will be called a ‘ground’ database;

\[
\forall r t a v. ([r, t, a, v]_\delta \Rightarrow \forall a' A(r, a') \Rightarrow \exists v' [r, t, a', v'_ \delta])
\]
Other, so called “integrity constraints” can be defined for a particular set of databases. Such a set of constraints $\chi$ is used by the predicate $C$ which is true iff a database satisfies the set of constraints:

$$(-C_\chi(\delta)) \text{ iff } (\chi \cup \delta) \supset \bot$$

If $Dom(\Gamma)$, then

$$\Delta_\Gamma$$

is the set of databases that are described by $\Gamma$, and

$$\Delta_{\Gamma,\chi}$$

is the subset of this where all the members satisfy the integrity constraints $\chi$.

In practise, the integrity constraints are expressed using the language of propositions in TRC. The constraints referred to above, are the interpretation of these in GRC.
Appendix B  The Semantics of TRC in terms of GRC

The meaning of TRC expressions can be defined in terms of GRC sentences. A model for TRC is a domain definition (data model) and an assignment function \( \llbracket . \rrbracket \), relativised to a database state \( \delta \) for assigning truth values to propositions in TRC\(^6\). There is a separate interpretation function \( I \) for interpreting TRC queries. \( g \) assigns values to the TRC constants. A model for TRC is thus

\[
\langle I_\delta, \llbracket . \rrbracket_\delta, \Gamma, g \rangle \text{ where } \text{Dom}(\Gamma) \text{ and } \delta \in \Delta_\Gamma
\]

The function \( I \) gives an interpretation of queries in TRC:

\[
I_\delta(\{ <t_1, a_1, \ldots, t_n, a_n> | t_1 \in r_1, \ldots, t_n \in r_n > | p \}) = A \text{ iff } \\
\exists v_1 \ldots v_n ( \\
\begin{array}{l}
[r_1, t_1, a_1, v_1]_\delta \land \ldots \land [r_n, t_n, a_n, v_n]_\delta \land \\
(\llbracket p [v_1/(t_1, a_1)] \ldots [v_n/(t_n, a_n)] \rrbracket_\delta \emptyset = e)
\end{array}
\land \\
A = \\
\emptyset
\]

\[
I_\delta(\{ <t_1, a_1, \ldots, t_n, a_n> | t_1 \in r_1, \ldots, t_n \in r_n > | p \}) = e \text{ iff } \\
\exists v_1 \ldots v_n ( \\
\begin{array}{l}
[r_1, t_1, a_1, v_1]_\delta \land \ldots \land [r_n, t_n, a_n, v_n]_\delta \land \\
(\llbracket p [v_1/(t_1, a_1)] \ldots [v_n/(t_n, a_n)] \rrbracket_\delta \emptyset = e)
\end{array}
\land \\
\emptyset
\]

The semantics of propositions in TRC are then:

\[
\llbracket \kappa \rrbracket_\delta \sigma = g(\kappa)
\]

\[
\llbracket t.a \rrbracket_\delta \sigma = v \text{ iff } [r, t, a, v]_\delta \land (t \in r) \in \sigma
\]

\[
\llbracket t.a \rrbracket_\delta \sigma = e \text{ iff } \neg A(r, a) \land (t \in r) \in \sigma
\]

\[
\llbracket \alpha = \alpha' \rrbracket_\delta \sigma = t \text{ iff } \\
(\llbracket \alpha \rrbracket_\delta \sigma = \llbracket \alpha' \rrbracket_\delta \sigma) \land \neg (\llbracket \alpha \rrbracket_\delta \sigma = e) \lor (\llbracket \alpha' \rrbracket_\delta \sigma = e)
\]

\[
\llbracket \exists i (T_i (\alpha) \land T_i (\alpha')) \rrbracket_\delta \sigma
\]

\[
\llbracket \alpha = \alpha' \rrbracket_\delta \sigma = e \text{ iff } \\
(\llbracket \alpha \rrbracket_\delta \sigma = e) \lor (\llbracket \alpha' \rrbracket_\delta \sigma = e) \lor \neg \exists i (T_i (\alpha) \land T_i (\alpha'))
\]

\[
\llbracket \alpha = \alpha' \rrbracket_\delta \sigma = f \text{ otherwise}
\]

---

6. It is assumed here that there are no null valued entries in the database.
\[ \| \alpha < \alpha' \|_\delta \sigma = t \text{ iff } \]
\[
(\| \alpha \|_\delta \sigma < \| \alpha' \|_\delta \sigma) \land \neg((\| \alpha \|_\delta \sigma = e) \lor (\| \alpha' \|_\delta \sigma = e)) \land \\
\exists i(T_i(\alpha) \land T_i(\alpha'))
\]
\[ \| \alpha < \alpha' \|_\delta \sigma = e \text{ iff } \]
\[
(\| \alpha \|_\delta \sigma = e) \lor (\| \alpha' \|_\delta \sigma = e) \lor \neg \exists i(T_i(\alpha) \land T_i(\alpha'))
\]
\[ \| \alpha < \alpha' \|_\delta \sigma = f \text{ otherwise } \]

\[ \| \overline{p} \|_\delta \sigma = t \text{ iff } \| p \|_\delta \sigma = f \]
\[ \| \overline{p} \|_\delta \sigma = e \text{ iff } \| p \|_\delta \sigma = e \]
\[ \| \overline{p} \|_\delta \sigma = f \text{ iff } \| p \|_\delta \sigma = t \]

\[ \| p \land p' \|_\delta \sigma = t \text{ iff } (\| p \|_\delta \sigma = t) \land (\| p' \|_\delta \sigma = t) \]
\[ \| p \land p' \|_\delta \sigma = e \text{ iff } (\| p \|_\delta \sigma = e) \lor (\| p' \|_\delta \sigma = e) \]
\[ \| p \land p' \|_\delta \sigma = f \text{ otherwise } \]

\[ \| p \lor p' \|_\delta \sigma = t \text{ iff } \]
\[
(\| p \|_\delta \sigma = t) \land \neg((\| p' \|_\delta \sigma = e) \lor (\| p' \|_\delta \sigma = t) \land \neg(\| p \|_\delta \sigma = e))
\]
\[ \| p \lor p' \|_\delta \sigma = e \text{ iff } (\| p \|_\delta \sigma = e) \lor (\| p' \|_\delta \sigma = e) \]
\[ \| p \lor p' \|_\delta \sigma = f \text{ otherwise } \]

\[ \| \text{All } t \in r. p \|_\delta \sigma = t \text{ iff } \]
\[
\forall t'([r, \hat{t}, a, v]_\delta \land \hat{v} \Rightarrow (\| p[t'/t] \|_\delta (\sigma \cup (t \in r)) = t))
\]
\[ \| \text{All } t \in r. p \|_\delta \sigma = e \text{ iff } \]
\[
\exists t'([r, t', a, v]_\delta \land \hat{v} \land (\| p[t'/t] \|_\delta (\sigma \cup (t \in r)) = e))
\]
\[ \| \text{All } t \in r. p \|_\delta \sigma = f \text{ otherwise } \]

\[ \| \text{Exists } t \in r. p \|_\delta \sigma = t \text{ iff } \]
\[
\exists t'([r, t', a, v]_\delta \land \hat{v} \land (\| p[t'/t] \|_\delta (\sigma \cup (t \in r)) = t))
\]
\[ \| \text{Exists } t \in r. p \|_\delta \sigma = e \text{ iff } \]
\[
\exists t'([r, t', a, v]_\delta \land \hat{v} \land (\| p[t'/t] \|_\delta (\sigma \cup (t \in r)) = e))
\]
\[ \| \text{Exists } t \in r. p \|_\delta \sigma = f \text{ otherwise } \]

The notion of “error” here is strong. If any sub-part of an expression is in error, or any part of the evaluation of a query is in error, then the whole expression is in error. It is possible to weaken this, so that, for example, a disjunction can be true if one disjunct is true, even if the other disjunct is in error (cf. “lazy” evaluation, and Kleene's strong and weak 3-valued connectives).
Appendix C  Proofs

Theorem 1

If $\delta \in \Delta$ and $(\| \varphi \|_\delta \emptyset = e)$ then for all $\delta' \in \Delta$ $(\| \varphi \|_{\delta'} \emptyset = e)$

Sketch Proof:

Base cases

$\| t . a \|_\delta \sigma = e$ iff $\neg A(r, a) \land (t \in r) \in \sigma$ Note that $A(r, a)$ is not dependent $\delta$.

$\| \alpha R \alpha' \|_\delta \sigma = e$ iff $(\| \alpha \|_\delta \sigma = e) \lor (\| \alpha' \|_\delta \sigma = e)$ (dealt with above), or,

$\neg \exists i (T_i(\alpha) \land T_i(\alpha'))$. Note that $T_i$ is not database dependent.

All other cases of error $e$ reduce to one of these two cases. Neither is database dependent.

Next, it is shown that if there is a set of database entries which can verify a TRC proposition, then the definitions given for forming updates to an existing database, and associated constraints, will find all such possibilities. The truth of a modal proposition then is reduced to finding one such update which is consistent with the existing constraints.

Theorem 2

If $\varphi$ is consistent with the current data-model (for any $\delta'$, $\| \varphi \|_{\delta} \neq e$) then for all database states $\delta$ then all $\chi_{U}(\varphi)$ consistent instantiations of null values in all $Update_U(\varphi)$ verify $\varphi$ (where $U \in Complete(\delta)$) for any update function $U_\delta$.

Sketch Proof:

We can try this:

1. For all updates $U$ (and database states $\delta$) that can be generated by all update functions of the form $U_\delta(\varphi)$, all groundings $U'$ of $U$ which are consistent with the constraints generated by the function $\chi_{U}(\varphi)$ verify $\varphi$. ($U'$ is a grounding of $U$ iff $\forall r \in \delta \subseteq \, \exists v \subseteq \delta$.)

2. Further, all $\chi_{U}(\varphi)$ consistent members of $Complete(\delta)$ also verify $\varphi$.

3. As $Clash_{\delta}(U)$ is only used to remove data from the database prior to the update information being added, it cannot affect the consequences of the update (it does however reduce the chance of the new database state being inconsistent).

Unpacking a TRC proposition directly, a sortal context $\sigma$ which keeps track of the relations that tuple variables range over is needed. If the update is produced by function $U_\delta$, then it should satisfy the following axioms:
The ground cases:

$$U(t. a = \kappa) \sigma = \{ [r, t, a, \kappa] \}$$

$$\chi(t. a = \kappa) \sigma = [r, t, a, v]_U \wedge v = \kappa^7$$

where $$(t \in r) \in \sigma$$

In this case $$U$$ is ground, and it is the only grounding. It clearly satisfies the constraint and verifies the proposition.

$$U(t. a = t'. a') \sigma = \{ [r, t, a, *], [r', t', a', *] \}$$

$$\chi(t. a = t'. a') \sigma = [r, t, a, v]_U \wedge [r', t', a', v']_U \wedge v = v'$$

where $$(t \in r) \in \sigma$$ and $$(t' \in r') \in \sigma$$

Any grounding of the update which satisfies the constraint clearly verifies the proposition. This is also the case for the last two ground instances.

$$U(t. a < \kappa) \sigma = \{ [r, t, a, *] \}$$

$$\chi(t. a < \kappa) \sigma = [r, t, a, v]_U \wedge v < \kappa$$

where $$(t \in r) \in \sigma$$

$$U(t. a < t'. a') \sigma = \{ [r, t, a, *], [r', t', a', *] \}$$

$$\chi(t. a < t'. a') \sigma = [r, t, a, v]_U \wedge [r', t', a', v']_U \wedge v < v'$$

where $$(t \in r) \in \sigma$$ and $$(t' \in r') \in \sigma$$

---

7. This is not essential
Driving in negation:

\[
\begin{align*}
U(t.a = \kappa) \sigma &= U(t.a \prec \kappa \lor \kappa < t.a) \sigma \\
\chi(t.a = \kappa) \sigma &= [r,t,a,v]_U \land v \neq \kappa \\
U(t.a = t'.a') \sigma &= U(t.a < t'.a' \lor t'.a' < t.a) \sigma \\
\chi(t.a = t'.a') \sigma &= [r,t,a,v]_U \land [r',t',a',v']_U \land v \neq v' \\
U(t.a < \kappa) \sigma &= U(\kappa < t.a) \sigma \\
\chi(t.a < \kappa) \sigma &= \chi(\kappa < t.a) \sigma \\
U(t.a < t'.a') \sigma &= U(t'.a' < t.a) \sigma \\
\chi(t.a < t'.a') \sigma &= \chi(t'.a' < t.a) \sigma \\
\end{align*}
\]

These re-express negation in terms of the ground cases, and so any instantiation of the update which satisfies the constraints verifies the proposition.

Decomposing the proposition:

\[
\begin{align*}
U(\square p) \sigma &= \emptyset \\
\chi(\square p) \sigma &= \emptyset \\
\end{align*}
\]

In this case the update is ground, and consistent! The modal proposition is dealt with when the whole proposition is evaluated with respect to the updated database. If we want to see whether the empty update verifies the proposition, then the expression \(\|\square p\|_\emptyset = t\) must be expanded. This will be verified in terms of the value of \(\|p\|_m\), where \(m\) is a \(\chi_U(p)\) valid instantiation of some update function \(U(p)\).

\[
\begin{align*}
U(p \Lambda p') \sigma &= U(p) \sigma \cup U(p') \sigma \\
\chi(p \Lambda p') \sigma &= \chi(p) \sigma \land \chi(p') \sigma \\
\end{align*}
\]

Now \(\|p \Lambda p'\|_U \sigma = \|p\|_U \sigma \land \|p'\|_U \sigma\), if all the instantiations of the conjuncts respective updates which are consistent with their respective constraints verify the conjuncts, then all the instantiations of the union of the updates, consistent with both sets of constraints, will verify the conjunction.

\[
\begin{align*}
U(p \lor p') \sigma &= U(p) \sigma \text{ or } U(p \lor p') \sigma = U(p') \sigma \\
\chi(p \lor p') \sigma &= \chi(p) \sigma \text{ iff } U(p \lor p') \sigma = U(p) \sigma \\
\chi(p \lor p') \sigma &= \chi(p') \sigma \text{ otherwise} \\
\end{align*}
\]
The argument for disjunction parallels that for conjunction, 
$\|p \lor p'|_{U}\sigma = \|p\|_{U}\sigma \lor \|p'|_{U}\sigma$: if all the instantiations of one of the updates, are consistent with the corresponding constraints, then all those instantiations will also verify the disjunction.

$$U(\text{Exists } t \in r.p)\sigma = U(p[t'/t]) (\sigma \cup \{t' \in r\}) \text{ for some } t'$$

$$\chi(\text{Exists } t \in r.p)\sigma = \chi(p[t'/t]) (\sigma \cup \{t \in r\}) \text{ iff }$$

$$U(\text{Exists } t \in r.p)\sigma = U(p[t'/t]) (\sigma \cup \{t' \in r\})$$

This is again reducing the proposition to a simpler case (in parallel with the semantics of TRC).

$$U(\text{All } t \in r.p)\sigma = \bigcup_{r':\exists\omega([r,r',a,v]_{\delta})} U(p[t'/t]) (\sigma \cup \{t' \in r\})$$

$$\chi(\text{All } t \in r.p)\sigma = \forall t' (\exists\omega([r,r',a,v]_{U}) \Rightarrow \chi(p[t'/t]) (\sigma \cup \{t' \in r\}))$$

Note that when adding these constraints to a database $\delta$, a global substitution of $[\delta/U]$ must be performed.

Now it must be shown that $\text{Complete}$ forms all the possible completions of an update, without affecting the verification of the proposition in question, if the completed updates are consistent with the constraints associated with the update.

$$\text{Complete}_{\delta}(U) =$$

$$\{S\}$$

$$\forall rt a\nu([r, t, a, v]_{U} \wedge \nu \Rightarrow [r, t, a, v]_{S})$$

All grounded entries in the update are members of all the completions.

$$\wedge$$

$$\forall rt a([r, t, a, *]_{U} \wedge \neg K(r, a) \Rightarrow [r, t, a, *]_{S}) \wedge$$

All null values in the update which are not keyfields, are in all completions

$$\forall rt a([r, t, a, *]_{U} \wedge K(r, a) \Rightarrow$$

$$([r, t, a, *]_{S} \lor \exists t' v'([r, t', a, v']_{\delta} \land [r, t, a, v']_{S}))$$

All null keyfield values either appear in the completion, or an existing keyfield value does.

$$\wedge$$

$$\forall rt a a' v'([r, t, a', v]_{U} \wedge A(r, a) \land \neg \exists v'.([r, t, a, v']_{U}) \Leftrightarrow$$

$$[r, t, a, *]_{S} \lor$$

$$\exists t' v'' (K(r, a) \land [r, t', a, v'']_{\delta} \land [r, t, a, v'']_{S})$$

\[\]
Any fields not mentioned are either filled in as null values, or, if a keyfield, they may adopt an existing keyfield value.

\[
\forall \forall r, t, a, v \exists r' ( [r, t, a, v]_S \land K(r, a) \land [r, t', a, v]_S \Rightarrow \\
\forall a' (\exists v'' ( [r, t, a', v'']_U \Rightarrow \\
\exists v' ( [r, t', a', v']_S \land [r, t, a', v']_S )) 
\]

For all keyfield values in the completed update which match the keyfields in the existing database, and have missing attribute values in the original (uncompleted) update, then all missing attribute values must be made equal to attribute values in the existing database (inheritance).

\[
\forall \forall v, v' ( [r, t, a, v]_S \land [r, t, a, v']_S \Rightarrow v = v') 
\]

There can only be one value for each attribute in a tuple.

So Complete essentially enriches an update to explicitly consider cases of inheritance from the existing database. Any new information (attribute values) cannot lead to a conflict with the constraints, because those attributes cannot appear in the constraints, if they do not appear in the update. Some of the instantiations of the null values may lead to a failure to satisfy the constraints associated with the update, but these would have failed when considering the uncompleted update.

Finally, considering the Clash, this is only used to remove conflicting information from the existing database, it cannot affect the propositions verified by the completions of the updates. When adding completed updates to the existing database, to produce new database states, information is added which is irrelevant to the proposition. Any entries which clash with the constraints associated with the update will have been removed.

The evaluation of the modal thus becomes a question of determining whether there is an instantiation of the updated database which satisfies both the constraints associated with the update, and existing constraints.

**Bibliography**

*Formal Semantics and Pragmatics for Natural Language Querying*
Cambridge University Press

*An Introduction to Database Systems*
Fifth Edition
Addison-Wesley

'A Natural Language System Based on Formal Semantics'
*Proceedings of the International Conference on Current Issues in Computational Linguistics (June 1991)*
Penang, Malaysia

'A Natural Language Front End To Relational Systems Based on Formal Semantics'
*Proceedings of InfoJapan'90 (October 1990)*
Tokyo, Japan

*Introduction to Montague Semantics*
D. Reidel

*Logic and Databases*
Plenum Press, New York

*An introduction to modal logic*
Methuen

*Introduction of Metamathematics*
Van Nostrand

'Logic and Databases'
*Logic Programming Meeting*
Imperial College London, May 1976
‘Naming and necessity’  
in Harman and Davidson (eds)  
*Semantics of Natural Language*  
Reidel 253-356  
Republished as  
*Naming & necessity* (1980)  
Basil Blackwell, Oxford

‘Constraint Checking for Modal Queries’  
*Computing Science Memorandum*  
University of Essex

‘Modal Reasoning in Relational Systems’  
*Computing Science Memorandum*  
University of Essex

‘Efficient Generation of SQL Expressions from Natural Language’  
in M. S. Jackson and A. E. Robinson (eds)  
*Aspects of Databases*  
(Proceedings of the Ninth British National Conference on Databases - July 1991)  
Butterworth-Heinemann, July 1991

‘Data Bases: Theory vs. Interpretation’  
in H. Gallaire and J. Minker (eds)  
*Logic and Databases*  
Plenum Press, New York

‘An Introduction to the Formal Specification of Relational Query Languages’  
The Computer Journal Vol. 28 No. 2

‘Question Answering’  
in S.C. Shapiro (ed)  
*Encyclopedia of Artificial Intelligence*  
Wiley, New York, 818-822