

QuabeC

Peter Seymour

2 March 2009

Last Updated: 15 July 2009

1 Introduction

QuabeC stands for *Querying and updating $a \rightsquigarrow b$ edges with Constraints* where $a \rightsquigarrow b$ refers to a general asymmetric 2-place relation: there is an edge from a to b .

The system provides a multi-versioned external storage mechanism. The underlying data structure is intentionally simple but more complicated structures can be built on top of it yielding a rich modelling platform.

2 Data Structure

Data is stored as sequences of graphs each consisting of directed edges over an infinitely countable set of nodes called atoms referred to by \mathbb{A} . Atoms are either named by an arbitrary length string over a fixed alphabet or are anonymous. There is a bijection between the set of all strings and the named atoms so that where an atom has a name it is unique amongst all the other names. Similarly there is a bijection between the anonymous atoms and the natural numbers. These two mappings induce a total order by placing anonymous atoms in order before the named atoms taken in lexicographic order. Each data graph is a model for a first-order language where the atoms are the domain and the edges define a relation. It is helpful to consider a concrete model of each graph as a subset of all the edges, written $G \subseteq \mathbb{A} \times \mathbb{A}$. Then in terms of the language:

$$G \models a \rightsquigarrow b \Leftrightarrow (a, b) \in G$$

It is important to note that it is the graph edges which define the state since the nodes are universally fixed across all graphs. A sequence of graphs represents an enumeration of its versions. Multiple sequences can be used to permit distinct datasets and branching. These will be written as $\emptyset, G_1, G_2, \dots$. Where \emptyset denotes the empty graph from which all other graphs will evolve.

3 Query Language

To describe this structure is an accompanying predicate calculus \mathcal{L}_Q which allows queries over the data structure. A query is a well-formed formula whose result depends on the number of free variables present. If there are no free variables in the query (a closed formula) then it is either **true** or **false**. If there is one free variable then the result is the set containing all atoms which satisfy the formula. Where there are two or more free variables the result is a set of tuples. The free variables in a formula are ordered so it is always possible to define such tuples.

3.1 Well-formed Formulæ

The alphabet of symbols is as follow:

$x, x_1, x_2, \dots; a, a_1, a_2, \dots; \hat{1}, \hat{2}, \dots; \dots$	variables
"a", "ab", ...	constants
$=, <, @^e, \Lambda_{a_1}^e, \Lambda_{a_2}^e, \dots; \phi^e, \phi_1^e, \phi_2^e, \dots; \varphi^e, \varphi_1^e, \varphi_2^e, \dots; \dots$	predicates
(,), ,	punctuation
\neg, \wedge	connectives
\forall	quantifier

Well formed formulæ (abbreviated to *wff*) are built-up from atomic formulæ using the connectives. Variables range over the atoms and occur in a *wff* either bound or free. Bound variables must fall under the scope of a quantifier. The free variables are labelled by \hat{n} to simplify binding rules. The constants denote specific named atoms.

3.1.1 Construction

A *term* is defined by:

1. A variable is a term.
2. A constant is a term.

An *atomic formula* is defined by:

1. If ϕ^e is a predicate and t_1, \dots, t_e are terms then $\phi^e(t_1, \dots, t_e)$ is an atomic formula.
2. If x and y are terms then $x = y$ is an atomic formula.
3. If x and y are terms then $x < y$ is an atomic formula.
4. If p and t_1, \dots, t_e are terms then $@^e(p, t_1, \dots, t_e)$ is an atomic formula.

A *wff* is defined by:

1. All atomic formulæ are *wff*s.
2. If \mathcal{A} and \mathcal{B} are *wff*s then $(\neg\mathcal{A}), (\mathcal{A} \wedge \mathcal{B})$ and $(\forall x)\mathcal{A}$ are *wff*s where x is any variable.
3. The set of *wff*s is generated from the above.

3.1.2 Custom Predicates

A *wff* can be considered as a predicate taking as many arguments as it has distinct free variables. If a *wff* has no free variables then it is closed. The superscript e on a predicate letter denotes the number of arguments that the predicate accepts. This can be thought of as the type of the predicate or as we shall see later the type of a set-abstract. For example:

$$\phi^2 = (\hat{1} \rightsquigarrow \hat{2}) \wedge (\hat{1} \rightsquigarrow \text{“quabec”})$$

The free variables can be bound by application as in:

$$\phi(x, y) = (x \rightsquigarrow y) \wedge (x \rightsquigarrow \text{“quabec”})$$

3.1.3 Atom Ordering

The pre-defined 2-place predicate $<$ orders the atoms such that anonymous atoms appear before named atoms and named atoms appear in lexicographic order. Anonymous atoms are unordered in \mathcal{L}_Q even though the model provides a total order. The equality predicate $=$ behaves as expected.

3.1.4 Interpreter

The interpreter predicate $@^e$ for $e \geq 0$ interprets the name of its first argument as a *wff* with the remaining e arguments (if any) bound to the free variables. This is always false for anonymous atoms, named atoms that do not denote a *wff* or misapplication of arguments.

3.2 Simultaneous Satisfaction

For each atom F there is a predicate Λ_F^e which uses F to define a range of atoms to be simultaneously interpreted. If F is a named atom whose name denotes a *wff* of exactly one free variable then Λ_F^e is evaluated as

$$\Lambda_{F(p)}^e(x^e) = (\forall p)(F(p) \rightarrow @^e(p, x^e))$$

Should F not be of the specified form the predicate is false.

3.2.1 Truth

Any closed *wff* constitutes a truth statement since it can be evaluated as true or false for any given model G .

$$G \models \phi$$

3.2.2 Set Abstraction

Reference to a subset of the atoms is achieved using a non-closed *wff*. Each combination of atoms satisfying the predicate forms a tuple. The full set of these tuples is the set-abstract.

$$S_{\phi^e} = \{x^e \mid \phi^e(x^e)\}$$

Where x^e is either a tuple of e elements or a single variable if $e = 1$. These set-abstracts are unordered in \mathcal{L}_Q but are ordered in the model.

3.3 Syntactic Extensions

The following set of definitions are syntactic and allow a more natural construction of *wff*. The use of the superscript e allows predicates and the set-abstracts they define to have a definite type. Certain pieces of syntax rely on types matching so variables can also be typed to make this explicit. This typing of variables is not part of the syntax so x^1 should be read as x and x^e as (x_1, \dots, x_e) . Where types are obvious they will be omitted.

To ease clarity predicates can be defined outside of any *wff*. When they are used in a *wff* they would be applied by substitution with all free variables replaced by bound terms. For instance:

$$\begin{aligned} \phi &= \neg(\hat{1} < \hat{2}) \\ \phi(\text{"a"}, \hat{1}) \end{aligned}$$

When defining a predicate the use of the enumerated free variables can be replaced by an explicit argument list. Similarly the result of the query is the set of all atom tuples satisfying the final *wff*. The above example can be re-written as:

$$\begin{aligned} \mathbf{pred} \phi(x, y) &= \neg(x < y) \\ &\{x \mid \phi(\text{"a"}, x)\} \end{aligned}$$

Variables can similarly be declared for simple substitution in the resulting *wff*.

$$\begin{aligned} \mathbf{var} t &= \text{"a"} \\ \mathbf{pred} \phi(x, y) &= \neg(x < y) \\ &\{x \mid \phi(t, x)\} \end{aligned}$$

Finally set-abstracts can be brought out.

```

var t = "a"
pred  $\phi(x, y) = \neg(x < y)$ 
set S = {x |  $\phi(t, x)$ }
S

```

3.3.1 Definitions

The following are to be taken as equivalent atomic formulæ:

$x \neq y$	\equiv	$\neg(x = y)$
$x > y$	\equiv	$y < x$
$x \leq y$	\equiv	$\neg(y < x)$
$x \geq y$	\equiv	$\neg(x < y)$
$x \sim y$	\equiv	$\neg(y < x) \wedge \neg(x < y)$
$x^e = y^e$	\equiv	$(x_1 = y_1) \wedge \dots$
$x^e \neq y^e$	\equiv	$(x_1 \neq y_1) \vee \dots$
$x^e < y^e$	\equiv	$(x_1 < y_1) \vee ((x_1 = y_1) \wedge ((x_2 < y_2) \vee \dots))$
$\phi^e(x^e)$	\equiv	$\phi^e(x_1, \dots, x_e)$
\top	\equiv	"0" < "1"
\perp	\equiv	$\neg\top$
$@p(x^e)$	\equiv	$@(p, x_1, \dots, x_e)$

The following are to be taken as equivalent *wffs*:

$\mathcal{A} \vee \mathcal{B}$	\equiv	$\neg(\neg\mathcal{A} \wedge \neg\mathcal{B})$
$\mathcal{A} \rightarrow \mathcal{B}$	\equiv	$\neg\mathcal{A} \vee \mathcal{B}$
$\mathcal{A} \leftarrow \mathcal{B}$	\equiv	$\mathcal{B} \rightarrow \mathcal{A}$
$\mathcal{A} \leftrightarrow \mathcal{B}$	\equiv	$(\mathcal{A} \rightarrow \mathcal{B}) \wedge (\mathcal{A} \leftarrow \mathcal{B})$
$(\exists x)\mathcal{A}$	\equiv	$\neg(\forall x)\neg\mathcal{A}$
$(\forall x^e)\mathcal{A}$	\equiv	$(\forall x_1) \dots (\forall x_e)\mathcal{A}$
$(\exists x^e)\mathcal{A}$	\equiv	$(\exists x_1) \dots (\exists x_e)\mathcal{A}$

The following are *wffs* defined by set-abstracts:

S_{ϕ^e}	\equiv	$\{x^e \phi(x^e)\}$
$x^e \in S_{\phi^e}$	\equiv	$\phi^e(x^e)$
$x^e \notin S_{\phi^e}$	\equiv	$\neg\phi^e(x^e)$
$S_{\phi^e} \subseteq S_{\varphi^e}$	\equiv	$(\forall x^e)(\phi^e \rightarrow \varphi^e)$
$S_{\phi^e} \supseteq S_{\varphi^e}$	\equiv	$(\forall x^e)(\phi^e \leftarrow \varphi^e)$
$S_{\phi^e} = S_{\varphi^e}$	\equiv	$(\forall x^e)(\phi^e \leftrightarrow \varphi^e)$
$S_{\phi^e} \cap S_{\varphi^e}$	\equiv	$\{x^e \phi^e \wedge \varphi^e\}$
$S_{\phi^e} \cup S_{\varphi^e}$	\equiv	$\{x^e \phi^e \vee \varphi^e\}$
$S_{\phi^e} \setminus S_{\varphi^e}$	\equiv	$\{x^e \phi^e \wedge \neg\varphi^e\}$
\emptyset^e	\equiv	$\{x^e \perp\}$
\mathbb{A}^e	\equiv	$\{x^e \top\}$
$\{a^e\}$	\equiv	$\{x^e x^e = a^e\}$
$\{a^e, b^e, \dots\}$	\equiv	$\{x^e x^e = a^e \vee x^e = b^e \vee \dots\}$

3.3.2 Reusable Queries

Consider a predicate that determines whether an atom is anonymous, $x < \text{“”}$.

```

var isanon = “ $\hat{1} < \text{“”}$ ”
set S = {x | @isanon(x)}
S

```

This works precisely because the atoms are immutable so the reference in the **var** clause can hold across queries.

Alternatively useful queries can be stored as part of the data structure, “*isanon*” \rightsquigarrow “ $\hat{1} < \text{“”}$ ”. Traditional databases offer views which can also be replicated by applying @ to an atom representing a set-abstract. The @ predicate acts as a very powerful interpreter of the query language.

In fact it can be argued that it is too powerful since it can effectively build uncomputable sets. This is the motivation for introducing simultaneous satisfaction as a weaker replacement. Any predicate determining formulæ to be interpreted will either generate a finite or infinite set. In the first case each case can be checked whereas in the latter the satisfaction must fail since there will always be one failing formula. If there were not a failing formula we would have computed an infinite set of valid formulæ in a single *wff* which cannot be the case.

4 Higher Order Relations

Since having only a 2-place relation is too restrictive for many uses we propose here a general n-place relation as a further extension. Given *a* and *b* such that $a \rightsquigarrow b$ we write $r : \langle a b \rangle \equiv a \rightsquigarrow r \wedge r \rightsquigarrow b$ and $\langle a b \rangle \equiv (\exists r)r : \langle a b \rangle$ for some atom *r*. The intention is to let *r* be anonymous and represent the relation between *a* and *b* so it can in turn be used within other relations. Taking this relation as a base we can specify high order relations as follows:

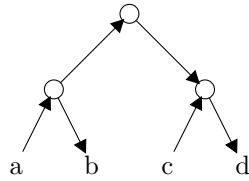
$$\begin{aligned}
r_n : \dots r_1 : \langle a_n \dots a_1 \rangle &\equiv r_n : \langle a_n r_{n-1} \rangle \wedge r_{n-1} : \dots r_1 : \langle a_{n-1} \dots a_1 \rangle \\
r_{n-1} : \dots r_1 : \langle a_n \dots a_1 \rangle &\equiv (\exists r_n) r_n : \dots r_1 : \langle a_n \dots a_1 \rangle \\
&\dots \\
\langle a_n \dots a_1 \rangle &\equiv (\exists r_n) \dots (\exists r_1) r_n : \dots r_1 : \langle a_n \dots a_1 \rangle
\end{aligned}$$

As a further extension whenever we see such a relation used in a position where an atom would be expected as in $\phi(r)$ then we write:

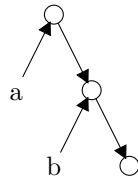
$$\phi(\langle a b \rangle) \equiv (\exists r) r : \langle a b \rangle \wedge \phi(r)$$

Analogous forms apply for nested structures.

One way to view this structure is as special form of tree over the atoms where fresh anonymous atoms are introduced at the nodes and the root node is the representative. Any tree structure can be modelled since there is no problem with nesting these n-place relations. As an example $\langle \langle a b \rangle \langle c d \rangle \rangle$ is perfectly legal.



The special syntax $\langle a b \dots \rangle$ represents a degenerate tree called a list.



5 Natural Form

Putting together all the pieces allows for a very natural data form. Each concept in the logical model should have an associated atom. Sometimes it makes sense to name the concept but in other cases a concept is more sensibly referenced by its position in the topology.

The concept of “mass” for instance can easily be named but an instance of a more general concept for example a particular person may not have a unique name. In the latter case an anonymous atom represents the concept and its

relations define it uniquely by imbuing it with properties such as age etc. This doesn't mean there is no concept of identity rather there is no named reference to it.

We can represent a class as an atom and have the concept of class membership as another. So we could use an atom "in" for membership and another for the class as in "People". Then for x to be a person the following must hold:

$$\langle x \text{ "in" "People"} \rangle$$

The membership concept is reusable as in $\langle x \text{ "in" "Cats"} \rangle$. We can enquire about all people using a set abstract $\{x | \langle x \text{ "in" "People"} \rangle\}$. People can have properties such as age $\langle x \text{ "hasage" "30"} \rangle$. It might be necessary to find all people under a certain age. This can be accomplished by establishing a numeric type within the current structure. If it is to be used in comparisons then it must obey $<$. This does not naturally occur but [1] gives a series of numeric types and composite types that will obey $<$. In which case assume $\overline{30}$ is in such a form then all young people is given by:

$$\{(x, n) | \langle x \text{ "in" "People"} \rangle \wedge \langle x \text{ "hasage" } n \rangle \wedge n < \overline{30}\}$$

It's worth noting that if $\langle a b c \rangle$ and $\langle a b \rangle$ hold then $\{x | \langle a x \rangle\} = \{\langle b c \rangle, b\}$. This logically makes sense when viewed as predicates on a but might not be intended. One way to differentiate is to see that $\langle b c \rangle$ is anonymous whereas b is possibly not.

6 Update Language

The query language views the data structure as static and produces no side effects. This is desirable especially when the @ predicate can mimic dynamic behaviour. However, at some point data needs to be created and changed. This is achieved using a simple update language. Since there are only atoms and edges it is relatively simple. If a concept is logically mutable it should be an atom in its own right referencing its state. For example moving from $\langle a \text{ "hasprop" } b_1 \rangle$ to $\langle a \text{ "hasprop" } b_2 \rangle$.

An update works on edges by specifying what should hold after the update has been made irrespective of what held before. This is accomplished in a similar manner to set abstraction except the selection is over relations *not* atoms. To produce an edge between two atoms we use the following primitive edge set statement.

$$[a \rightsquigarrow b | \top]$$

Similarly to remove a relation $[\neg a \rightsquigarrow b | \top]$. As a language extension $[\phi | \top]$ can be written as $[\phi]$. The statement on the right-hand side is any *wff* from \mathcal{L}_Q . The statement on the left-hand side is composed of atoms, \wedge , \neg , edge sets and anonymous-selection. Any of the extensions to \mathcal{L}_Q that use these components

may also be considered valid. The interpretation of \wedge is that both statements to its left and right will be satisfied after the update.

The result of an update is an edge set containing edges that should be present after the update and those that should not. This could be in conflict e.g. $[a \rightsquigarrow b \wedge \neg a \rightsquigarrow b]$. In this instance the update fails and no change will be observed. An update statement will not directly change the underlying data model but forms the first of two distinct phases. The first determines the edge set on a static view of the data whereas the second enforces this edge set by producing a change set which will remove existing edges and create edges that don't already exist.

Consider $[\neg(a \rightsquigarrow b \wedge b \rightsquigarrow a)]$. Without the negation this is only satisfied by adding the two relations. In its negative sense either or both relation could be removed and the statement will hold after the update. Since making a choice would introduce indeterminism into the update language both are chosen. Therefore it simply reverses the satisfaction of each component, $\neg a \rightsquigarrow b \wedge \neg b \rightsquigarrow a$. Notice that this implies the former and is fully determined. We can now interpret $p \vee q$ as $\neg(\neg p \wedge \neg q)$ which conforms to its use in the query language. An unfortunate consequence is that certain logically correct updates will not be deterministic and hence not permitted.

$$[a \rightsquigarrow b \vee \neg a \rightsquigarrow b]$$

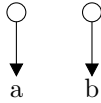
Anonymous-selection is a powerful technique to add relations in a declarative way. All anonymous atoms can only be referenced through a query that looks into the atom's local topology. To introduce such an atom requires constructing that topology on a fresh atom. An anonymous selection takes the form of a quantifier $(+r)\phi$ where ϕ is any valid left-hand side statement. It is to be interpreted that an anonymous atom r will be selected such that it satisfies $\neg(\exists t)(r \rightsquigarrow t \vee t \rightsquigarrow r)$ before the update and is distinct from all other such selections. It can be considered a fresh atom and since the anonymous atoms are well-ordered this selection process can always be made.

We now look at two standard uses for the quantifier. This is best seen using an example shown as a graph. Initially there will be two atoms a and b with no edges between them.

a b

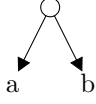
The first update associates to each of them a new anonymous atom of their own.

$$[(+r)(r \rightsquigarrow x) | x \in \{a, b\}]$$

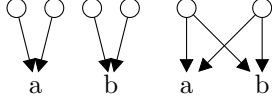


The second update associates to each of them a new anonymous atom shared between them.

$$[(+r)[r \rightsquigarrow x | x \in \{a, b\}]]$$



Further applications cause more new atoms to be associated.



To produce idempotent updates the right-hand side of the edge set can be used to query for the desired relation.

$$[(+r)(r \rightsquigarrow a) | \neg(\exists t)(t \rightsquigarrow a)]$$

To generalise this we extend the syntax with a new quantifier $(*r)$ such that the following equivalences hold:

$$\begin{aligned} [(*r)\phi(r) | \varphi] &\equiv [(+r)\phi(r) | \varphi \wedge \neg(\exists t)\phi(t)] \\ [(*r)[\phi(r) | \varphi]] &\equiv (+r)[\phi(r) | \varphi \wedge \neg(\exists t)\phi(t)] \end{aligned}$$

This all leads up to some tidy definitions concerning the higher-order relations. Suppose we wish to assert $\langle a \ b \rangle$ and $\langle a \ b \ c \rangle$ respectively then the following updates achieve them:

$$[(*r)(r : \langle a \ b \rangle)] \equiv [(*r)(a \rightsquigarrow r \wedge r \rightsquigarrow b)]$$

$$[(*r_1)(*r_2)(r_1 : r_2 : \langle a \ b \ c \rangle)] \equiv [(*r_1)(*r_2)(a \rightsquigarrow r_1 \wedge r_1 \rightsquigarrow r_2 \wedge b \rightsquigarrow r_2 \wedge r_2 \rightsquigarrow c)]$$

This can be more consisely written if we are not interested in using some of the new atoms representing the relations.

$$\begin{aligned} [* \langle a \ b \rangle] &\equiv [(*r)(r : \langle a \ b \rangle)] \\ [* \langle a \ b \ c \rangle] &\equiv [(*r_1)(*r_2)(r_1 : r_2 : \langle a \ b \ c \rangle)] \end{aligned}$$

7 Graph Evolution

This sections looks at how a graph evolves when updates are applied. All queries are against a specific version of a specific graph. We begin by considering each step along a graph sequence. Then we can can build from here to look at the relationship between sequences and applying updates concurrently.

7.1 Physical Change Sets

The state of the model at any sequence point t is simply the set G_t of edges that exist between atom pairs. An update takes the form (U_t^+, U_t^-) , where an edge in U_t^+ is to exist after an update, an edge U_t^- is to not exist and all the rest are to remain unaltered. A *change set* on the other hand determines how G_{t+1} is to be constructed from G_t by toggling the membership for certain edges. We therefore represent a change set as $\Delta_t \subseteq \mathbb{A}^2$ such that:

$$\Delta_t = (U_t^+ \cap \overline{G_t}) \cup (U_t^- \cap G_t)$$

See appendix A for notation. The following then holds:

$$G_{t+1} = (G_t \cup \Delta_t) \setminus (G_t \cap \Delta_t) = G_t \oplus \Delta_t$$

The properties of the change set operator \oplus are also detailed in appendix A.

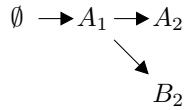
A sequence of change sets $\Delta_0 \dots \Delta_{t-1}$ fully determines G_t by induction if it is assumed that $G_0 = \emptyset$. Therefore:

$$G_t = \bigoplus_{0 \leq i < t} \Delta_i$$

These are referred to as the *physical* change sets because they take a graph through its successive versions one at a time building on the previous. Since they fully determine G_t we can view each graph sequence as its sequence of change sets $\langle \Delta_0, \dots, \Delta_t, \dots \rangle$.

7.2 Branching

All graphs grow from a common root \emptyset but they can also branch at later points. Consider $\langle \Delta_0^A, \Delta_1^A, \dots \rangle$ and $\langle \Delta_0^A, \Delta_1^B, \dots \rangle$ as change sets for graph sequences of $\langle A_i \rangle$ and $\langle B_i \rangle$ respectively. As shown in the diagram the graph sequence $\langle B_i \rangle$ inherits its first change from $\langle A_i \rangle$ (it branches from A at version 1).



We can say $\Delta_0^B = \Delta_0^A$ and more generally if B branches from A at t then $\Delta_i^B = \Delta_i^A \forall i < t$. To express this branching directly we write:

$$\begin{aligned}
A_0 &= \emptyset \quad [\text{branch point}] \\
A_1 &= A_0 \oplus \Delta_0^A \\
A_2 &= A_1 \oplus \Delta_1^A
\end{aligned}$$

$$\begin{aligned}
B_0 &= A_0 \\
B_1 &= A_1 \quad [\text{branch point}] \\
B_2 &= B_1 \oplus \Delta_1^B
\end{aligned}$$

Since \emptyset is a common root for all graphs we can talk about *the graph tree* which considers all branches (graph sequences) as part of a single structure.

7.3 Merging

Given a graph G_t consider Δ_t^A and Δ_t^B which are physical change sets applied to G_t to form two branches A_{t+1} and B_{t+1} . Under what circumstances can these branches be merged to produce G_{t+1} ? It is desirable that the results of changes occurring to G_t by Δ_t^A and Δ_t^B be jointly preserved in G_{t+1} . Without loss of generality we apply Δ_t^A first and then ensure that those edges which will be changed by Δ_t^B have not already been altered by Δ_t^A . This is expressed as follows using the change set restriction operator $G \upharpoonright_X = (G \cap X, \overline{G} \cap X)$:

$$(G_t \oplus \Delta_t^A) \upharpoonright_{\Delta_t^B} = G_t \upharpoonright_{\Delta_t^B}$$

This necessitates two conditions

$$\begin{aligned}
(G_t \oplus \Delta_t^A) \cap \Delta_t^B &= G_t \cap \Delta_t^B && \text{(by definition)} \\
(G_t \cap \Delta_t^B) \oplus (\Delta_t^A \cap \Delta_t^B) &= G_t \cap \Delta_t^B && \text{(by } (\Delta 5)) \\
\Delta_t^A \cap \Delta_t^B &= \emptyset && \text{(apply } \oplus(G_t \cap \Delta_t^B))
\end{aligned}$$

and

$$\begin{aligned}
\overline{(G_t \oplus \Delta_t^A)} \cap \Delta_t^B &= \overline{G_t} \cap \Delta_t^B && \text{(by definition)} \\
\overline{(G_t \oplus \Delta_t^A)} \cap \Delta_t^B &= \overline{G_t} \cap \Delta_t^B && \text{(by } (\Delta 2)) \\
\overline{(G_t \oplus \Delta_t^B)} \cap (\Delta_t^A \cap \Delta_t^B) &= \overline{G_t} \cap \Delta_t^B && \text{(by } (\Delta 5)) \\
\Delta_t^A \cap \Delta_t^B &= \emptyset && \text{(apply } \oplus(G_t \cap \Delta_t^B))
\end{aligned}$$

So we can proceed to define a change set merge operator \uplus such that:

$$X \uplus Y = X \oplus Y \text{ where } X \cap Y = \emptyset$$

otherwise a conflict occurs and no merging can proceed.

As an example consider a graph with a two edges “a” \rightsquigarrow “1” and “b” \rightsquigarrow “1”. Then the change sets $\{(\text{“a”}, \text{“1”}), (\text{“a”}, \text{“2”})\}$ and $\{(\text{“b”}, \text{“1”}), (\text{“b”}, \text{“2”})\}$ can be merged without conflict and increment the two counters. However, the change sets $\{(\text{“a”}, \text{“1”}), (\text{“a”}, \text{“2”})\}$ and $\{(\text{“a”}, \text{“2”}), (\text{“a”}, \text{“3”})\}$ can be applied in sequence but would cause a merge conflict.

It should be noted that even if a merge conflict doesn’t arise that consistency could still be violated since the edges involved in the query part of the update are not checked for conflict. This would occur if one update relies on reading (but not changing) an edge that is then changed by another change set. These two change sets could still be merged but certain properties that exist after each taken in isolation may not be preserved. So it can be said that the merge operator handles write conflicts but not read conflicts. Indeed this is true of all the change set operators, they handle the physical aspects of the graph tree evolution. Any logical aspects should be enforced by constraints which are discussed later.

7.4 Logical Change Sets

From the building blocks of physical changes sets we can look at *logical change sets* which consist of one or more physical changes sets rolled up. For instance:

$$\delta_{t,t+k} = \Delta_t \oplus \dots \oplus \Delta_{t+k-1}$$

These can be treated just like physical change sets with regard to merging and branching. Notice that changes can occur during a logical change set that would normally conflict in merging but so long as these changes are later reversed within that logical change set no conflict will occur.

The generalised merge rule is:

$$G_{t+1} = G_t \oplus (\delta_{t,t+k_1}^1 \uplus \dots \uplus \delta_{t,t+k_n}^n)$$

Despite each branch moving multiple versions, the merged branch is only advanced once since only one change set is applied to it.

7.5 Grafting

So far merging of branches has been shown to be possible where a common branch point is identified. However, it may be the case that a section of a graph sequence represented as a logical change set needs to be applied to another branch where no common root can be established at the point of the change set. Since change sets are not idempotent it is necessary to carry over some of the original graph structure to ensure the change set is correctly applied. Consider

two graph branches G_r and H_s where a common root only exists strictly before $\min(r, s)$ (remembering that all branches have \emptyset as a common root). Then $\delta_{s,s+k}^H$ can be applied to G_r only if the following condition holds:

$$G_r \upharpoonright_{\delta_{s,s+k}^H} = H_s \upharpoonright_{\delta_{s,s+k}^H}$$

This ensures the change set is applied under the same conditions on both branches. It trivially holds where $r = s$ and the branching is coming from a common root so merging is a special case. Since $\delta_{s,s+k}^H$ is recoverable from $H_s \upharpoonright_{\delta_{s,s+k}^H}$ (see appendix A) only the latter is necessary for the definition of the graft. We can write a change set grafting operator $G_r \searrow X$ as follows:

$$G_r \searrow H_s \upharpoonright_{\delta_{s,s+k}^H} = G_r \oplus \delta_{s,s+k}^H \text{ where } G_r \upharpoonright_{\delta_{s,s+k}^H} = H_s \upharpoonright_{\delta_{s,s+k}^H}$$

7.6 Undo

This simple case is handled by reapplying the last change set to *undo* its effects. In symbols this amounts to $G \oplus X \oplus X = G$.

8 Serialisation Strategies

The previous section covered all the necessary algebra to extensively handle branching, merging and grafting in a very general manner. However, at no point was there any consideration for the overall evolution of the graph tree in real time. This would be the case for any realisation of QuabeC as a database system.

The abstraction we use in this section is that of a sequence of requests arriving at the graph tree. Simultaneous requests are arbitrarily ordered to simplify the model. We write G_r^{read} and G_w^{write} as the read node and write node respectively for a request. For a query request there is simply the read node and a query formula. There is no issue of conflict with multiple query requests so long as their read nodes exist.

An update request requires a query request and a write node to apply the resulting change set to. This write node must be a leaf in the graph tree so it can be immediately applied. Given that grafting is possible the read node and write node are independent so long as the grafting conditions hold.

The next stage is to develop an algorithm for determining the read and write nodes for incoming requests. It is not feasible for a requesting system to know precisely which nodes it needs since it may require the ‘latest’ or ‘next’ node on a given branch. We label each branch with a simple name where there are as many branches as there are leaf nodes. This means that non-leaf nodes are not uniquely named since each branch name labels all nodes from the leaf to the root. The algorithm works with a triple of data (G^{read}, k, G^{write}) .

$$(G^{read}, k, G^{write}) \rightarrow \begin{cases} (G_k^{read}, G_w^{write}) & 0 < k \leq r \\ (G_r^{read}, G_w^{write}) & k = 0 \\ (G_{w+k}^{read}, G_w^{write}) & -w \leq k < 0 \\ failure & \text{otherwise} \end{cases}$$

Where w is the next available write sequence number on G^{write} and r is largest sequence number on G^{read} . If G^{write} does not exist it is created as a new branch such that $G_r^{write} = G_r^{read}$ and $w = r + 1$. If updates are handled serially then $w = r + 1$ but introducing the notion that updates can be occurring concurrently allows for a wider variety of interpretations. For instance $(G, -1, G)$ is a serial update on G that waits for all current updates to finish (G_{w-1}^{read} may not yet exist although it will be scheduled for creation). A non-waiting update would take the form $(G, 0, G)$ and a simple branch $(G, 0, G')$ where G' does not exist. If $G' \neq G$ exists then the last update would be a graft.

The algorithm is both lock-free and wait-free since a natural order is assigned to each update and no update can prevent another finishing for any more than a finite period. This is a direct consequence of updates being of finite duration.

8.1 Long Transactions

Long transactions can be implemented in a non-blocking way by breaking them into smaller updates. Each part would specify the same fixed read node and rely on merging to detect write conflicts.

A greater degree of security can be obtained by implementing a locking or versioning mechanism within the graph structure. The simplest example is a counter. By ensuring that the long transaction is the sole incremter of a given counter it can be sure of its own consistency. However, in the face of failure it would have to implement its own rollback procedure. This may not always be possible but is an essential problem whose only alternative is to lock some section of the graph at the risk it is never released. Note that this strategy can also be enforced so long as all updates go through a single coordinator which is a suggested strategy for using QuabeC. See section 11 for more details.

9 An Alternative Representation

The underlying graph is potentially countably infinite. Consider $[r \rightsquigarrow "x"]$, this will associate each atom with "x" creating one edge per atom. However, viewed another way the graph structure is always determined by a finite number of finite update formulæ. So there is at least one finite representation of any potential graph, namely, the sequence of updates creating it.

We now look at how to construct queries and updates as solutions to satisfaction problems. To simplify this construction we associate each atom with an element of \mathbb{Z} such that the anonymous atoms are all negative and each named atom retains its lexicographic order under the usual integer order. This means "" becomes 0 and to disambiguate the old ordering we write $<_{\mathbb{A}}$. This forces an

order on the anonymous atoms which simplifies the following constructions. In this simpler model a graph is a subset of $\mathbb{Z} \times \mathbb{Z}$.

9.1 Queries

A query is an expression of the form $\{(x_1, \dots, x_n) | \phi\}$ where ϕ has n free variables. If ϕ has m quantifiers then its bound variables are x_{n+1}, \dots, x_{n+m} all distinct from each other.

9.1.1 Reduction

We first recursively re-write ϕ according to the following rules to bring it into prenex disjunctive normal form.

- Reduce ϕ to its canonical form by removing all uses of extensions.
- Replace each occurrence of $\neg(\forall x_i)(\phi)$ with $(\exists x_i)(\neg\phi)$.
- Replace each occurrence of $\neg(\exists x_i)(\phi)$ with $(\forall x_i)(\neg\phi)$.
- Replace each occurrence of $\neg\neg\phi$ with ϕ .
- Replace each occurrence of $x <_{\mathbb{A}} y$ with $x < y \wedge \neg(y < 0)$.
- Replace each occurrence of $\neg(x = y)$ with $(x < y) \vee (y < x)$.
- Replace each occurrence of $\neg(x < y)$ with $(y < x) \vee (x = y)$.
- Replace each occurrence of $\neg(x \rightsquigarrow y)$ with $x \not\rightsquigarrow y$.
- Replace each occurrence of $(\phi \vee \varphi) \wedge \psi$ with $(\phi \wedge \psi) \vee (\varphi \wedge \psi)$.
- Replace each occurrence of $\neg(\phi \wedge \varphi)$ with $\neg\phi \vee \neg\varphi$.
- Replace each occurrence of $\neg(\phi \vee \varphi)$ with $\neg\phi \wedge \neg\varphi$.
- Replace each occurrence of $\Lambda_{F(p)}^e(x^e)$ depending on F :
 - F is not a valid 1 place predicate, replace by \perp .
 - F is satisfied by an infinite set of atoms, replace by \perp .
 - F is satisfied by known p_1, \dots, p_n , replace by $@p_1(x^e) \wedge \dots \wedge @p_n(x^e)$.

The resulting query takes the form:

$$\{(x_1, \dots, x_n) | (Q_{n+1} \dots Q_{n+m})(C_1 \vee \dots \vee C_r)\}$$

Where each Q_i is either \forall or \exists and there are no occurrences of \neg . Each C_i is the conjunction of atomic formulæ of x_i or integer constants involving $<, =, \rightsquigarrow$ or $\not\rightsquigarrow$ only.

In this first step we assume the query is over $G_0 = \emptyset$ so a further reduction can be done such that all occurrences of $x \not\rightsquigarrow y$ are removed and any C_i with an

occurrence of $x \rightsquigarrow y$ is completely removed. We now have a fixed model over $\mathbb{Z} \times \mathbb{Z}$ which will form the basis for arbitrary updates.

The next step involves building a graph for each conjunctive clause C as follows:

1. Create an equivalence class $X = \{x_i\}$ for each x_i .
2. For each occurrence of $x < y$ in C assign an edge from X containing x to Y containing y . (X and Y maybe the same).
3. For each occurrence of $x = y$ combine X containing x to Y containing y to form a single class.

Call this graph R which may not be fully connected. If any class in R contains two differing constants or if R contains a cycle then C is false and can be removed entirely.

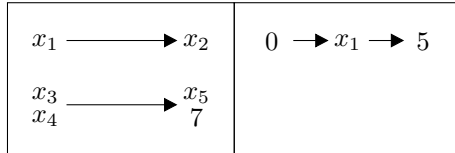
We now proceed to define lower and upper bounds lb and ub on each class X in R .

$$lb(X) = \begin{cases} \kappa - \frac{1}{2} & \kappa \in \mathbb{Z} \\ \min(\{lb(Y) | Y \rightarrow X\}) & \\ -\infty & \text{otherwise} \end{cases}$$

$$ub(X) = \begin{cases} \kappa + \frac{1}{2} & \kappa \in \mathbb{Z} \\ \max(\{ub(Y) | X \rightarrow Y\}) & \\ \infty & \text{otherwise} \end{cases}$$

This is sufficient to generate all tuples of the inner expression as follows. The tuples satisfying each C_i are the cartesian product of all the sub-tuples generated by each connected subgraph. Variables occurring within any X are bounded in $(lb(X), ub(X))$. They are also constrained by variables in classes leading from and to that class as well as variables and constants in their own class. The total tuple set is the union of the tuples generated by each C_i .

For example the following generates tuples $(x_1, x_2, x_3, x_4, 7)$ such that $x_1 < x_2$ and $x_3 = x_4 < 7$ along with $(x_1, x_2, x_3, x_4, x_5)$ such that $0 < x_1 < 5$.



9.1.2 Quantification

Given the above graph construction we can see that within any conjunction term C a given variable x is bounded within a known interval. We aim to remove the innermost quantifier.

We proceed by forming sub-intervals $\{J_j\}$ of \mathbb{Z} for x by taking the end points of all intervals which bound x across all the graphs along with $-\infty$ and ∞ . Each interval I is then the union of some subset of the $\{J_j\}$. Suppose $I = J_1 \cup J_2$ then:

$$\phi = x \in I \wedge \phi = (x \in J_1 \wedge \phi) \vee (x \in J_2 \wedge \phi)$$

If for some J there is no I that contains it we include:

$$x \in J \wedge x < x$$

To keep the graph representation in accordance with this re-write new variants will need to be formed. For each graph find the class X containing x and produce variants for each $J \subseteq I$ by including constant bounds. This requires adding $\{lb(J)\} \rightarrow X \rightarrow \{ub(J)\}$. Note that constants can appear in multiple equivalence classes without harm.

We handle the two sorts of quantifier differently.

Universal Quantification

Consider a simple case $(\forall x)((x \in J_1 \wedge \phi_1(x, y)) \vee (x \in J_2 \wedge \phi_2(x, y)))$ such that $\{J_1, J_2\}$ is a partition of \mathbb{Z} . The aim is to generate all y that satisfy this formula. This is equivalent to the following chain of formulæ:

$$\begin{aligned} & (\forall x)((x \in J_1 \vee x \in J_2) \wedge (x \in J_2 \vee \phi_1) \wedge \\ & \quad (x \in J_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_2)) && \text{expansion} \\ & (\forall x)((x \in J_2 \vee \phi_1) \wedge (x \in J_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_2)) && J_1 \cup J_2 = \mathbb{Z} \\ & (\forall x)((x \in J_1 \rightarrow \phi_1(x, y)) \wedge (x \in J_2 \rightarrow \phi_2(x, y)) \wedge (\phi_1 \vee \phi_2)) && \mathbb{Z} \setminus J_1 = J_2 \\ & (\forall x)(x \in J_1 \rightarrow \phi_1(x, y)) \wedge (\forall x)(x \in J_2 \rightarrow \phi_2(x, y)) && \text{implication} \end{aligned}$$

This result can be extended for more than two terms. To solve for y we look at the constraints on ϕ_1 when $x \in I_1$ simultaneously with the analogous case for ϕ_2 .

For each J_j there will be one or more corresponding graph variants produced above. We consider each possible combination such that there is exactly one graph variant for each J_j . Universal quantification requires satisfaction across all of \mathbb{Z} . For each combination the identity above is used to combine the graph variants and remove x . Then the disjunction of all combinations is taken as the result. Each graph is pre-processed in effect calculating $(\forall x)(x \in J \rightarrow \phi)$.

- Step 1
Find the class X containing x and ensure $X = \{x\}$ (or if it contains a constant then $J = \{x\}$).
- Step 2
For each Y such that $Y \rightarrow X$ add $Y \rightarrow \{lb(X)\}$.
- Step 3
For each Y such that $X \rightarrow Y$ add $\{ub(X)\} \rightarrow Y$.

Across all variants merge all classes which share members and bring over all edges. Cycles and conflicts with distinct constants appearing in the same class are resolved by skipping the affected combination (there may be not valid combinations). Lower and upper bounds are now re-calculated and any empty intervals also result in the affected combination being skipped.

Existential Quantification

This situation is easier. Again consider a simple case $(\exists x)((x \in J_1 \wedge \phi_1(x, y)) \vee (x \in J_2 \wedge \phi_2(x, y)))$ but this time J_1 and J_2 do not need to form a partition, they could even be equal. This is equivalent to:

$$(\exists x)(x \in J_1 \wedge \phi_1(x, y)) \vee (\exists x)(x \in J_2 \wedge \phi_2(x, y))$$

Each graph is processed as follows but no new disjunction needs to be taken.

- Step 1
Find the class X containing x .
- Step 2
For each Y such that $Y \rightarrow X$ add $Y \rightarrow \{ub(X)\}$.
- Step 3
For each Y such that $X \rightarrow Y$ add $\{lb(X)\} \rightarrow Y$.

Cycles and conflicts can't occur but empty intervals result in the graph being removed from the disjunction.

9.1.3 Resulting Structure

The final situation is a graph representation in the same form as before but with all references to x resolved. The method is repeated until all quantifiers are resolved arriving at a graph representation which can generate all tuples.

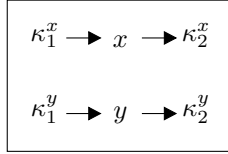
9.2 Updates

Take an update of the form $[x \rightsquigarrow y|\phi]$ this is calculated as a query $\{(x, y)|\phi\}$. Similarly $[\neg x \rightsquigarrow y|\phi]$ uses the same query. Any update $[R_1 \wedge R_2|\phi]$ can use the union of queries for each component update. Updates are a restricted form of queries where only two variables occur. We can now study updates in this context. The graph representation used for queries reveals enough information to fully describe updates as queries over two variables x and y . Conjunctions are handled as the union of the resulting subsets of $\mathbb{Z} \times \mathbb{Z}$. Anonymous selection is handled by preselecting a batch of atoms and re-writing the formulæ.

Any graph with only x , y and constants appearing can only take one of the forms described next.

9.2.1 Unrelated

Each of x and y appear in their own sub-graph. They can only have edges to and from classes containing a single constant and/or be in a class with a constant. If they appear in a class with a constant the constant can be removed and bounding classes used in place. If no bounds appear then dummy bounds of $-\infty$ or ∞ can be introduced. The only form remaining is:



9.2.2 Related

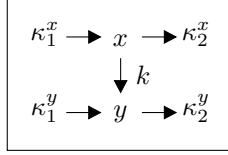
In the second case we have x and y related by atomic formulæ.

- $x < \dots < y$
- $y < \dots < x$
- $x = y$

The dots arise since quantified variables could have left empty classes on paths between x and y . Should constants appear on such paths then x and y are really unrelated and fall into the first case. Define an edge label k by:

$$k = \begin{cases} n & x < z_1 < \dots < z_{n-1} < y \\ 0 & x = y \\ -n & y < z_1 < \dots < z_{n-1} < x \end{cases}$$

There will also be constant bounds as before. This results in the following graph:



9.3 G_1

Now updates over $G_0 = \emptyset$ have been covered the form of G_1 is the union of graphs or their complements following 2 simple forms. From each of these forms it is easy to reconstruct a *wff* that represents them. These *wff*s are the logical form of G_1 that can be substituted into queries whenever $x_i \rightsquigarrow x_j$ or $x_i \not\rightsquigarrow x_j$ occurs. The preceding method of handling queries then carries over to subsequent graph updates.

This alternative representation demonstrates how a graph tree can remain finite.

10 Constraint Atoms

The flexibility of the data model poses a risk that the structure will become unmanageable and hard to understand. To prevent this a series of constraints can be imposed which will define a data schema. A constraint is represented by a truth statement held as an atom's name. Only one atom needs to be proposed since it can combine the results of other constraint atoms. Let c_1, \dots, c_n be atoms whose names are valid truth statements posing as constraints. Then choose C such that:

$$\mathbf{var} C = "c_1 \wedge \dots \wedge c_n"$$

The constraint atom C will be used as a check to ensure consistency is maintained. This method is simple but any constraint changes will require proposing a new constraint atom. By using a $\Lambda_{F(p)}$ predicate this can be avoided. Suppose that for any constraint atom c the relation $\langle c \text{ "in" "Constraints"} \rangle$ holds. Then C can be chosen such that:

$$\begin{aligned} \mathbf{var} F &= "\langle \hat{1} \text{ "in" "Constraints"} \rangle" \\ \mathbf{var} C &= "\Lambda_{F(c)}(c)" \end{aligned}$$

Should any constraint atom not contain a valid statement then the constraint fails. This prevents invalid constraints being formed since any update that introduces an invalid constraint will be undone. A constraint atom is specified as part of any update. Where the constraint fails the update is not applied but the sequence number on the branch is still incremented so as not to break the serialisation algorithm.

11 High-Level Usage

As presented QuabeC should be considered as a physical backend for a rich database system. It provides all the mechanisms required for extremely complex data management and evolution that encompasses common functions of databases and version control systems simultaneously. However, it doesn't provide certain features such as security, locking, mirroring, backup, etc that maybe desired. All of these features can be built on top of QuabeC and are discussed here.

Each graph tree is considered to be an individual "database" and would have a dedicated co-ordinator responsible for assigning permissions to updates and queries. The co-ordinator's role is to act as a single point of interaction with the graph tree by client systems. As requests arrive they can be re-directed to storage engines that manage subsets of the nodes on the tree. In this way it is possible to parallelise the system and prevent bottlenecks. To mirror the tree is paramount to sending each update request to an alternative instance in the same order they are processed in the master system. The co-ordinator will be responsible for bundling constraints with requests to any branch since there maybe security concerns at this level.

Archiving maybe appropriate and can be managed by the coordinator. Pruning the tree from the root maintains the branch structure but prevents certain branch nodes being available. The co-ordinator can report errors arising from accessing non-existent nodes as it would if future nodes were referenced.

12 Implementation

This section sketches some ideas on implementation. The state of the graph tree is uniquely defined by the sequence of updates that lead up to its current structure. This can be used for physical storage. First create a string b-tree to map referenced atom names to a natural number. Anonymous atoms can be numbered more easily. Take each update and replace all atom references by the assigned number. Store both the pool of atom names and each update formula. As an update is committed entries are written to the string pool and then the update appended to the update log.

The database would be inefficient to handle like this so an in memory representation is kept that is efficient to handle. Where this exceeds the memory bounds it can be paged to external storage as blocks. It doesn't matter whether blocks are flushed often since the definitive tree state is held by the update log. On failure this can be consulted to repair the running representation. The blocks may contain some portion of an uncommitted update at any time without harm.

It's not necessary to prematurely force blocks to external storage. By deferring this steps enables multiple blocks to be written without additional seeks. Crucially an update could incur as little cost as the single commit to the update log requiring only a single seek if it contains no new atom names.

If archiving occurs a check point of the running representation at the archive point can be made, the update log truncated and the string pool garbage collected. A second consideration is branching the update log but this is harder where grafting occurs.

The running representation will take the form of blocks handling various edges. Edges are not necessarily treated in isolation but in sets are discussed in section 9. For each edge set it is only necessary to record at what version it was toggled. At a branch point a link is made to its parent branch. When blocks are overfull the edge sets can spill into additional blocks such that like versions are held as closely together as possible. Searching back to the root for each query counting the number of times edges have been toggled can be avoided by keeping a status flag in each block for some known version.

A Δ -algebra

Some results about the Δ -algebra are given here. We write \overline{X} to mean the complement of X in \mathbb{A}^2 .

The definition of the change set combine operator \oplus is such that:

$$X \oplus Y = (X \cup Y) \setminus (X \cap Y)$$

By definition we can see that $X \oplus Y = Y \oplus X$ and $X \oplus X = \emptyset$.

$$\begin{aligned} X \oplus Y &= (X \cup Y) \setminus (X \cap Y) \\ &= (X \cup Y) \cap \overline{(X \cap Y)} \\ &= (X \cup Y) \cap (\overline{X} \cup \overline{Y}) && (\Delta 1) \\ &= (X \cap \overline{Y}) \cup (\overline{X} \cap Y) && (\Delta 2) \end{aligned}$$

$$\begin{aligned} \overline{X \oplus Y} &= \overline{(X \cup Y) \cap (\overline{X} \cup \overline{Y})} && (\text{by } (\Delta 1)) \\ &= \overline{(X \cup Y)} \cup \overline{(\overline{X} \cup \overline{Y})} \\ &= (\overline{X} \cap \overline{Y}) \cup (X \cap Y) && (\Delta 3) \\ &= (\overline{X} \cup Y) \cap (X \cup \overline{Y}) \\ &= \overline{(\overline{X} \cup Y) \cap (\overline{X} \cup Y)} \\ &= \overline{\overline{X} \oplus Y} = X \oplus \overline{Y} && (\Delta 4) \end{aligned}$$

$$\begin{aligned}
(X \oplus Y) \oplus Z &= ((X \oplus Y) \cap \overline{Z}) \cup (\overline{(X \oplus Y)} \cap Z) && \text{(by } \Delta 2) \\
&= ((X \cap \overline{Y}) \cup (\overline{X} \cap Y) \cap \overline{Z}) \cup && \text{(by } \Delta 2) \\
&\quad ((\overline{X} \cap \overline{Y}) \cup (X \cap Y) \cap Z) && \text{(by } \Delta 3) \\
&= (X \cap \overline{Y} \cap \overline{Z}) \cup (\overline{X} \cap Y \cap \overline{Z}) \cup \\
&\quad (\overline{X} \cap \overline{Y} \cap Z) \cup (X \cap Y \cap Z) \\
&= X \oplus (Y \oplus Z) && \text{(by symmetry)}
\end{aligned}$$

$$\begin{aligned}
(X \oplus Y) \cap Z &= (X \cup Y) \cap \overline{(X \cap Y)} \cap Z \\
&= ((X \cup Y) \cap Z) \cap ((\overline{X} \cup \overline{Y}) \cup \overline{Z}) && (Z \cap \overline{Z} = \emptyset) \\
&= ((X \cap Z) \cup (Y \cap Z)) \cap ((\overline{X} \cup \overline{Z}) \cup (\overline{Y} \cup \overline{Z})) \\
&= ((X \cap Z) \cup (Y \cap Z)) \cap (\overline{(X \cap Z)} \cup \overline{(Y \cap Z)}) \\
&= ((X \cap Z) \cup (Y \cap Z)) \cap \overline{((X \cap Z) \cap (Y \cap Z))} \\
&= (X \cap Z) \oplus (Y \cap Z) && (\Delta 5)
\end{aligned}$$

The definition of the change set restriction operator \downarrow_X is such that:

$$G \downarrow_X = (G \cap X, \overline{G} \cap X)$$

If $G \downarrow_X = (A, B)$ then $X = A \cup B$.

B Unresolved Questions

- Anonymous atoms could be exhausted by updates such as $[x \rightsquigarrow x]$. Should certain updates be denied?
- Can atoms be *typed* such that x^t implies $x \rightsquigarrow t$? eg. $\{x^t | \phi(x)\} \equiv \{x | x \rightsquigarrow t \wedge \phi(x)\}$ and $(\forall x^t) \phi(x) \equiv (\forall x)(x \rightsquigarrow t \rightarrow \phi(x))$. This would force all graphs to be necessarily finite since atom ranges would be explicitly defined by their *type*.
- Could anonymous selection be made a right-hand side term?
- Do the merge conflict conditions work in all relevant situations? Certainly they do in transitions such as from $a \rightsquigarrow 1$ to $a \rightsquigarrow 2$ since it contains both a removal and an addition.

References

- [1] “Efficient Lexicographic Encoding of Numbers”, Peter Seymour.