INTRODUCTION TO TRANSACTION LOGIC

— TUTORIAL —

Michael Kifer
University at Stony Brook
N.Y. 11794, U.S.A.

* Transaction Logic was developed jointly with Tony Bonner of University of Toronto
History

- 1991: decided to look into the theoretical foundations of logic programming with updates
- 1992: serial Transaction Logic is born
- 1994: graduates to concurrent Transaction Logic
- 1995: Transaction F-logic
- 1996: serial part of Transaction Logic implemented
- 1997: an implementation of Transaction F-logic (in Spain)
- 1998: (forthcoming) more efficient implementation of Transaction Logic
What Transaction Logic Is

- A logic designed for programming state-changing actions, executing them, and reasoning about their effects
- General logic, a conservative extension of classical predicate calculus
- Integrates declarative queries, transactional updates (abort, rollback, nested transactions), and composition thereof in one uniform, logical framework
- General Model Theory
  - Can do monotonic and non-monotonic reasoning
  - We do not want to commit to a particular choice of a non-monotonic theory: *Let’s first understand the logic behind the phenomenon of updates!* Well-founded, stable, etc., semantics are orthogonal issues
- Proof Theory
  - Sound and complete
  - SLD-style for so-called serial-Horn programs (a generalization of the regular Horn programs)
What Transaction Logic Is (contd)

- Makes **no assumption** about the nature of the database states being updated. A database state can be:
  - relational databases
  - disjunctive databases
  - logic programs
  - classical first-order theories
  - non-logical entities
- Makes **no assumptions** about the nature of elementary updates, which can be:
  - simple tuple insertion/deletions
  - relational SQL-style bulk updates
  - updates/revisions of logical theories
  - non-logical state changes done by an algorithm
- **But**: if assumptions are made, Transaction Logic can be used to *reason* about the effects of actions
What Transaction Logic Is Not

- Not another theory of updates for another logical theory
  - not an attempt to explain what “update $\phi$ with $\chi$” means
  - but such theories can be adapted/developed/used
- Not another variation on the theme of the situation calculus
- Not of Datalog-With-A-State-Argument variety
Why Transaction Logic?

- No acceptable logical language where transactional updates are integrated with queries and have a clean, logical semantics.
- No acceptable logical account for methods with side effects in object-oriented languages.
- No logic of action became the basis for updates in databases or logic programming.

Contrast with:
Classical logic is a basis for queries in logic programming and databases.
What Transaction Logic Does

Logic:
- transactional assert/retract
- methods in object-oriented DBMS
- integration of declarative and "procedural" knowledge

Transactional features:
- nested transactions
- atomicity
- isolation
- triggers
- deterministic and non-deterministic transactions
- dynamic constraints
What Transaction Logic Does (contd)

Control:
- subroutines
- serial and parallel composition of processes
- recursion, loops, conditionals
- communication and synchronization between concurrent processes

AI:
- logic for specifying and reasoning about actions
- language for specifying and generating plans
- frame problem:
  - not an issue for action execution
  - much smaller issue for reasoning about actions
• Path: \( \pi = \langle s_1, s_2, s_3, s_4, s_5, s_6 \rangle \)
• Real world and semantics: \( \phi \) executes along \( \pi \) \( \equiv \) \( \phi \) is true on \( \pi \)
• Proof theory: executes \( \phi \) along \( \pi \) as it proves \( \phi \)
Syntax

\( \wedge, \vee, \neg \) — “classical” connectives

\( \otimes, |, \odot, \diamond \) — new connectives

- \( \alpha \wedge \beta \) — execute \( \alpha \) so that it would also be a valid execution of \( \beta \).
  (Usually used in the context where \( \beta \) is a constraint on the execution of \( \alpha \).)

- \( \alpha \vee \beta \) — execute \( \alpha \) or execute \( \beta \) (non-determinism).

- \( \neg \alpha \) — execute in any way, provided that the resulting execution is not a valid execution of \( \alpha \).

- \( \alpha \otimes \beta \) — Execute \( \alpha \) then execute \( \beta \) (serial conjunction).

- \( \alpha | \beta \) — Execute \( \alpha \) and \( \beta \) in parallel (parallel conjunction).

- \( \odot \alpha \) — Execute \( \alpha \) in isolation (like in the database theory).

- \( \diamond \alpha \) — Check if execution of \( \alpha \) is possible.

- \( \exists X \alpha(X) \) — Execute \( \alpha \) for some \( X \).
Syntax: examples

Rules:

- \( a \leftarrow b \) \( (\equiv \ a \lor \neg b) \) means: one way to execute \( a \) is to execute \( b \).
  Operationally: subroutine definition. \( E.g., \)
  \[ a \leftarrow b \otimes (c \mid d) \otimes e \]
  \[ a \leftarrow f \otimes ((g \otimes h) \mid \circ (k \otimes f)) \]
  \[ a \leftarrow \diamond p \otimes q \otimes r \]

Read: \( a \) is a subroutine, which can be executed in one of the following three ways:

1. execute \( b \), then \( c \) and \( d \) concurrently, then \( e \); or
2. execute \( f \), then execute \( g \) followed by \( h \) concurrently with an isolated execution of \( k \) followed by \( f \); or
3. check if executing \( p \) is possible; if so, execute \( q \) then \( r \)

Constraints:

- \( p \land (\text{path} \otimes a \otimes \text{path}) \), where \( \text{path} \equiv \phi \lor \neg \phi \) - Transaction Logic’s “true”, means: execute \( p \) in such a way that action \( a \) is executed at some point during the process

- \( p \land \neg(\text{path} \otimes a \otimes \text{path}) \), means: execute \( p \) in such a way that action \( a \) is never executed in the process

- \( p \land \neg(\text{path} \otimes a \otimes \neg b \otimes \text{path}) \), means: execute \( p \) so that if \( a \) is executed at some point, then \( b \) is executed right after that
Overview of the Semantics

Any formula in Transaction Logic is a transaction/action/updating program/... (formulas with high degree of indeterminacy are better thought of as dynamic constraints, though).

- Formulas (i.e., transactions) have truth values and execution paths.
- Truth (or falsehood) is always over paths, not over states.

• A path is a sequence of states.
• Transaction $\phi$ being true on path $\pi = \langle s_1, s_2, s_3, ..., s_n \rangle$ means:
  $$\phi$$ can execute at state $s_1$, changing it to state $s_2$, ..., to $s_n$, terminating at $s_n$.
  $$\Rightarrow \text{ Truth over a path } \equiv \text{ execution over that path.}$$

• There is more to it with parallel execution. Basic idea: execution happens over multi-paths — paths with “pauses”; other transactions can execute during those pauses.

Queries are transactions that execute over 1-paths (length-1; have the form $\langle s \rangle$).
$$\Rightarrow$$ queries are transactions that do not change state.

• When execution is restricted to 1-paths, Transaction Logic reduces to classical logic
• The three conjunctions, $\land$, $\otimes$, $|$, then all reduce to the classical $\land$,
• but they are distinct notions over $n$-paths ($n > 1$).
Examples of Execution

- Let \( \phi_1 = a.del \otimes b.ins \otimes d \otimes c.ins \) 
  
  (\( a.del, b.ins, d, \) etc., are propositions for now, will explain later) 
  
  \( \phi_1 \) started at state \( \{d, a\} \) can pass through states \( \{d\}, \{d, b\} \); verifies that \( d \) is true at the latter state; then goes to state \( \{d, b, c\} \), and terminates. 
  
  \( \Rightarrow \) \( \phi_1 \) is true over path \( \pi = \langle \{d, a\}, \{d\}, \{d, b\}, \{d, b, c\} \rangle \).

- Let \( \phi_2 = \phi_1 \otimes e \) 
  
  Works like \( \phi_1 \), but at the end (at \( \{d, b, c\} \)) checks if \( e \) is true. 
  
  Finds out that \( e \) is false, so \( \pi \) is not an execution path of \( \phi_2 \). 
  
  \( \Rightarrow \phi_2 \) is false over \( \pi \). 
  
  (In fact, it happens to be false over every path that starts at \( \{d, a\} \) in some model.)

What is the nature of states? 
And what are these strange-looking symbols: \( a.del, b.ins \), etc.?
States

• Can think of the states as sets of atoms.
• Or formulas.
• But this is inadequate, in general:
  \[ p \leftarrow q \] means one thing in classical semantics, another in logic programming.
  Throw in the stable-model vs. well-founded semantics, add some spice
  (disjunctive programs, stationary semantics), and you get the idea.
• Transaction logic \textit{isolates} the details of state semantics from the rest through data oracles:
  – A \textbf{data oracle} is simply a mapping
    \[ \mathcal{O}^d : States \rightarrow Sets \ of \ First\text{-}order \ Formulas \]
  – \( \mathcal{O}^d(s) \) tells the logic what’s true at state \( s \).
Elementary Updates

- The strange-looking *a.del*, *b.ins*, etc., are just some ordinary propositions that happen to denote elementary updates (merely our notational convention).
- The semantics of elementary updates is specified via *transition oracles*.
- Transaction Logic is parameterized by data oracles and transition oracles.
- Each incarnation of the logic has its own data oracle (determines the set of allowed states and their semantics) and transition oracle (determines the set of allowed elementary transitions).
- The rest of the logic is independent of this choice: once the oracles are specified, the machinery cranks up and begins to run.
**Transition Oracles**

- *Transition oracles* are mappings of the form:
  \[ \mathcal{O}^t : \text{States} \times \text{States} \rightarrow \text{SetsOfGroundAtoms} \]

- \( b \in \mathcal{O}^t(D_1, D_2) \) means, executing \( b \) causes state transition from state \( D_1 \) to \( D_2 \).

**In this tutorial:** States are relational databases (sets of atoms).

State transitions can be of only these kinds:

- **Insert:** \( p.\text{ins}(t_1, \ldots, t_n) \in \mathcal{O}^t(D_1, D_2) \) iff \( D_2 = D_1 \cup \{p(t_1, \ldots, t_n)\} \).

- **Delete:** \( p.\text{del}(t_1, \ldots, t_n) \in \mathcal{O}^t(D_1, D_2) \) iff \( D_2 = D_1 \setminus \{p(t_1, \ldots, t_n)\} \).

Can have more complex elementary updates: theory revision/update a la Katsuno-Mendelzon, rule insertion/deletion to/from logic programs, stack operations, etc.
A Database Example: Financial Transactions

\begin{align*}
\text{transfer}(\text{Amt}, \text{Acct1}, \text{Acct2}) & \leftarrow \text{withdraw}(\text{Amt}, \text{Acct1}) \mid \text{deposit}(\text{Amt}, \text{Acct2}) \\
\text{withdraw}(\text{Amt}, \text{Acct}) & \leftarrow \circ (\text{balance}(\text{Acct}, \text{Bal}) \otimes \text{Bal} \geq \text{Amt} \\
& \quad \otimes \text{changeBalance}(\text{Acct}, \text{Bal}, \text{Bal} - \text{Amt})) \\
\text{deposit}(\text{Amt}, \text{Acct}) & \leftarrow \circ (\text{balance}(\text{Acct}, \text{Bal}) \otimes \text{changeBalance}(\text{Acct}, \text{Bal}, \text{Bal} + \text{Amt})) \\
\text{changeBalance}(\text{Acct}, \text{Bal1}, \text{Bal2}) & \leftarrow \text{balance.del}(\text{Acct}, \text{Bal1}) \\
& \quad \otimes \text{balance.ins}(\text{Acct}, \text{Bal2})
\end{align*}

- All variables are implicitly universally quantified (as usual in LP).

**Query:**

\begin{align*}
? - \text{transfer}(\text{Fee}, \text{Client}, \text{Broker}) \mid \text{transfer}(\text{Cost}, \text{Client}, \text{Seller})
\end{align*}

- Note: Prolog will **not** execute correctly anything analogous to this (because actions in Prolog lack transactional features).
**Semantics  Path Structures**

A *path structure* is a creature that assigns ordinary first-order semantic structures to paths (more precisely, multi-paths, but we will not press this issue here):

\[
M : \text{Paths} \rightarrow \text{FirstOrderSemanticStructures}.
\]

Two conditions tie in the oracles:

- **Data oracle compliance**: if \( D \) is a state, \( \phi \) is a first-order formula, and \( \mathcal{O}^d(D) \models^c \phi \) (\( \models^c \) means classical logical entailment), then \( M(\langle D \rangle) \models^c \phi \).

- **Transition oracle compliance**: If \( \mathcal{O}^t(D_1, D_2) \models^c \psi \) then \( M(\langle D_1, D_2 \rangle) \models^c \psi \).

Omitting some gory details:

1. **Base Case**: \( M, \pi \models p(t_1, \ldots, t_n) \) iff \( M(\pi) \models^c p(t_1, \ldots, t_n) \),
   
   for any atomic formula \( p(t_1, \ldots, t_n) \).
   
   (Read: \( p(t_1, \ldots, t_n) \) is a query or a transaction invocation; \( \pi \) is its execution path)

2. **Negation**: \( M, \pi \models \neg \phi \) iff \( \text{not}(M, \pi \models \phi) \).
   
   (Read: cannot execute \( \phi \) along the path \( \pi \).)
3. **“Classical” Conjunction:** $M, \pi \models \phi \land \psi$ iff $M, \pi \models \phi$ and $M, \pi \models \psi$.
(Read: can exec $\phi$ and $\psi$ along the same path—dynamic constraints.)

4. **Serial Conjunction:** $M, \pi \models \phi \otimes \psi$ iff $M, \pi_1 \models \phi$ and $M, \pi_2 \models \psi$
for some paths $\pi_1, \pi_2$ such that $\pi = \pi_1 \circ \pi_2$. (Read: do $\phi$ then $\psi$.)
Semantics — Path Structures (contd.)

5. **Concurrent Conjunction**: \( M, \pi \models \phi \mid \psi \) iff \( M, \pi_1 \models \phi \) and \( M, \pi_2 \models \psi \) for some paths \( \pi_1, \pi_2 \) such that \( \pi \in \pi_1 \parallel \pi_2 \).
(Read: do \( \phi \) and \( \psi \) concurrently.)

\[
\begin{array}{c}
\phi \\
\hline
\psi \\
\hline
\phi \\
\hline
\psi \\
\hline
\phi \\
\hline
\psi \\
\hline
\phi \\
\hline
\psi \\
\hline
\phi \\
\hline
\psi \\
\hline
\phi \\
\hline
\psi \\
\hline
\phi \\
\hline
\psi
\end{array}
\]

6. **Possibility**: \( M, \langle s_1 \rangle \models \Diamond \phi \) iff there is a path \( \pi = \langle s_1, \ldots, s_n \rangle \) such that \( M, \pi \models \phi \).

Note: \( \Diamond \phi \) is always a query (is true at states, even if \( \phi \) executes over a sequence of states longer than 1).

- Will not properly define \( \odot, \mid, \exists \) in this tutorial (so read!)
Semantics – Example

So, why is $\phi_1 = a.\text{del} \otimes b.\text{ins} \otimes d \otimes c.\text{ins}$ true (executes) over the path $\pi = \langle \{d, a\}, \{d\}, \{d, b\}, \{d, b, c\} \rangle$?

Let $M$ be a path structure. By the definitions of our oracle and path structures:

- $O^t(\{d, a\}, \{d'\}) \models a.\text{del}$, hence $M, \langle \{d, a\}, \{d\} \rangle \models a.\text{del}$
  
  $\{d, a\} \xrightarrow{a.\text{del}} \{d\}$

- $O^t(\{d\}, \{d, b\}) \models b.\text{ins}$, hence $M, \langle \{d\}, \{d, b\} \rangle \models b.\text{ins}$
  
  $\{d\} \xrightarrow{b.\text{ins}} \{d, b\}$

- $O^d(\{d, b\}) \models d$, hence $M, \langle \{d, b\} \rangle \models d$

- $O^t(\{d, b\}, \{d, b, c\}) \models c.\text{ins}$, hence $M, \langle \{d, b\}, \{d, b, c\} \rangle \models c.\text{ins}$
  
  $\{d, b\} \xrightarrow{c.\text{ins}} \{d, b, c\}$

$\Rightarrow$ the definition of $\otimes$ implies that then $M, \pi \models \phi_1$
More generally, let $P = \{ p \leftarrow a.del \otimes b.ins \otimes d \otimes c.ins \}$ (a transaction program). As before: $\pi = \langle \{d, a\}, \{d\}, \{d, b\}, \{d, b, c\} \rangle$.

We can show that if $M$ is a path structure where $P$ is true over every path, then also

$$M, \pi \models p$$

In fact, $M, \pi \models a.del \otimes b.ins \otimes d \otimes c.ins$ implies $M, \pi \models p$ in such path structures. Read: $P$ defines the subroutine $p$.

Are there $M$'s where the above is not true? — No!

In contrast, in some path structures $M_1, \pi \not\models a.del \otimes b.ins \otimes d \otimes c.ins \otimes e$ and in some $M_2, \pi \models a.del \otimes b.ins \otimes d \otimes c.ins \otimes e$

- This leads to the notion of executional entailment.
Execution as Logical Entailment

Let $P$ be a transaction program — a bunch of formulas (transaction definitions).

- $M$ is a model of $P$ iff $M, \pi \models \phi$ for every path $\pi$ and every $\phi \in P$.
- If $\phi$ is a formula, and $D_0, D_1, \ldots, D_n$ is a sequence of database state ids, then executional entailment is a statement of the form:
  \[
P, D_0, D_1, \ldots, D_n \models \phi
  \]

It means:

\[
M, \langle D_0, D_1, \ldots, D_n \rangle \models \phi
\]

for every model $M$ of $P$. 
Proof Theory

A simple SLD-style procedure for Concurrent Horn Clauses.

Just 4 inference rules:

- An SLD-like rule.
- A rule for dealing with queries to states.
- A rule for executing state transitions.
- A rule for isolated execution.

Concurrent Horn Clauses:

- Rules of the form: \( \text{atom} \leftarrow \text{ConcurrentSerialGoal} \)
- Concurrent Serial Goal:
  - An atomic formula; or
  - \( \phi_1 \otimes \ldots \otimes \phi_k \), where each \( \phi_i \) is a concurrent serial goal; or
  - \( \phi_1 | \ldots | \phi_k \), where each \( \phi_i \) is a concurrent serial goal; or
  - \( \bigodot \phi \), where \( \phi \) is a concurrent serial goal.
Proof Theory (contd.)

- Uses **sequents** of the form: \( P, D \vdash \phi \)
  meaning: \( \phi \) can execute starting from state \( D \), given the transaction definitions in \( P \).

- Inference rules are of the form: \( \text{Condition}, \quad \frac{\text{sequent}_1}{\text{sequent}_2} \)
  meaning: if \( \text{Condition} \) is true and \( \text{sequent}_1 \) has been proven then derive \( \text{sequent}_2 \).

- Proves statements of the form: \( P, D \vdash \phi \)
  and finds the execution path along the way.

**Axiom:** \( P, D \vdash () \)
where \( () \) is the *empty* concurrent serial goal.
Proof Theory Example

A top-down proof of \( \mathbf{P}, \{ c, d \} \vdash p \mid (a \otimes c.del \otimes d.del) \) where \( \mathbf{P} = \{ p \leftarrow a.ins \otimes b.ins \} \).

\[
\begin{align*}
\mathbf{P}, \{ c, d \} & \vdash (a.ins \otimes b.ins) \mid (a \otimes c.del \otimes d.del) \\
\mathbf{P}, \{ c, d, a \} & \vdash b.ins \mid (a \otimes c.del \otimes d.del) \\
\mathbf{P}, \{ c, d, a \} & \vdash b.ins \mid (c.del \otimes d.del) \\
\mathbf{P}, \{ d, a \} & \vdash b.ins \mid d.del \\
\mathbf{P}, \{ a \} & \vdash b.ins \\
\mathbf{P}, \{ a, b \} & \vdash ()
\end{align*}
\]

Ended up with an axiom \( \Rightarrow \) done!

Extract execution path from the proof:
\[
\{ c, d \}, \{ c, d, a \}, \{ d, a \}, \{ a \}, \{ a, b \}
\]

Final state: \( \{ a, b \} \).
Proof Theory—Inference rules

- No variables, to simplify exposition.

1. Applying transaction definitions: Let \( b \leftarrow \beta \in P \).

\[
P, D \vdash (\beta \otimes \alpha) \mid \gamma
\]

\[
P, D \vdash (b \otimes \alpha) \mid \gamma
\]

2. Querying the database: If \( \mathcal{O}^d(D) \models^c d \):

\[
P, D \vdash \alpha \mid \beta
\]

\[
P, D \vdash (d \otimes \alpha) \mid \beta
\]

3. Executing elementary updates: If \( \mathcal{O}^t(D_1, D_2) \models^c u \):

\[
P, D_2 \vdash \alpha \mid \beta
\]

\[
P, D_1 \vdash (u \otimes \alpha) \mid \beta
\]

4. Isolated execution of transactions:

\[
P, D \vdash \alpha \otimes (\beta \mid \gamma)
\]

\[
P, D \vdash (\circ (\alpha \otimes \beta)) \mid \gamma
\]
More Examples: Blocks World

\[ \text{stack}(N, X) \leftarrow N > 0 \otimes \text{move}(Y, X) \otimes \text{stack}(N - 1, Y) \]
\[ \text{stack}(0, X) \leftarrow \]
\[ \text{move}(X, Y) \leftarrow \text{pickup}(X) \otimes \text{putdown}(X, Y) \]

\[ \text{pickup}(X) \leftarrow \text{clear}(X) \otimes \text{on}(X, Y) \otimes \text{on.del}(X, Y) \otimes \text{clear.ins}(Y) \]
\[ \text{putdown}(X, Y) \leftarrow \text{wider}(Y, X) \otimes \text{clear}(Y) \otimes \text{on.ins}(X, Y) \otimes \text{clear.del}(Y) \]

Note: \text{stack} is non-deterministic.

Can go beyond specification of actions: it is easy to declaratively specify a planning strategy (e.g., STRIPS), crank the proof theory and out comes a plan!
Summary

- A logic for specifying, executing, and reasoning about transactions.
- Syntax:
  - *Serial logic*: first-order plus $\otimes$, $\Diamond$
  - *Concurrent logic*: serial plus $\mid$, $\circ$.
- Parameterized by data and transition oracles
  Can “plug in” different oracles and get different logics, tailored to specific applications.
- Model theory, proof theory.
- Uniformly integrates queries, updates, and transactions.
Applications

1. Transactional updates in logic programming and deductive databases.
2. Active databases.
3. Consistency maintenance.
4. Hypothetical reasoning.
5. Planning.
6. Object-oriented databases.
7. Workflow management systems.

All for the price of (1)!
Further Info

One implementation of the serial part of Transition Logic, one more forthcoming.

Tony Bonner maintains a page at

http://www.cs.toronto.edu/~bonner/transaction-logic.html

I also maintain related info at

http://www.cs.sunysb.edu/~kifer/dood/

(will put this tutorial there soon).