A Calculus for Reactive Databases

JOXAN JAFFAR, LIMSOON WONG AND ROLAND H.C. YAP

School of Computing
National University of Singapore
Singapore 119260
Republic of Singapore

{joxan,ryap}@comp.nus.edu.sg limsoon@krdl.org.sg

07 Jan 2000

Abstract

Traditionally, interaction with a database is done using a query language like SQL. Such queries apply only to the current state of the database, that is, a query entails computation on a static database. Further, queries are independent of each other in the sense that there is no direct mechanism for defining cooperative behavior. Increasingly, applications are required to live and operate over *updates* to the database. They are often required to respond, in some timely fashion, to updates. This dynamic behavior this requires that the application is *reactive* in nature. Further, it is becoming more important that applications can be designed *cooperate* with one another explicitly.

In this paper, we are concerned with how to "systematically react" to updates in a programmable way. While there is existing work in active databases, the programming language and concurrency interactions at the user level are largely unexplored. We present an abstract framework for writing programmable reactivity within a shared store which has suitable structure. There are potentially many agents accessing and updating this store. We put particular emphasis on this shared store being a database. This framework provides a simple reactor programming language which combines ideas from databases with process calculi. Such reactors can be embedded within arbitrary concurrent agents (processes) to write applications which are intended to react to changes, synchronize with each other, perform updates atomically and provide transactional behavior similar to traditional databases.

Contact Author:
Roland Yap
School of Computing, National University of Singapore
S16, Level 5, Room 05/08, 3 Science Drive 2,
Singapore 117543, Republic of Singapore

Email: ryap@comp.nus.edu.sg
Phone: +65 874-2972, Fax: +65 779-4580

1 Introduction

Traditionally, interaction with a database is done using a query language like SQL. Such queries apply only to the current state of the database, that is, a query entails computation on a static database. Further, queries are independent of each other in the sense that there is no direct mechanism for defining cooperative behavior. Increasingly, applications are required to live and operate over *updates* to the database. They are often required to respond, in some timely fashion, to updates. This dynamic behavior this requires that the application is *reactive* in nature. Further, it is becoming more important that applications can be designed *cooperate* with one another explicitly. Examples of applications which need reactivity are ERP systems, workflow systems, monitoring systems, agents in E-commerce settings, etc.

In this paper, we present an abstract framework, the $\mathcal{R}$-calculus, for the definition of database applications with the following key properties:

- It is a general-purpose concurrency calculus;
- Communication is founded upon notions about a shared store, and further, the store can be modelled as a database;
- It has explicit mechanisms for concurrency control, the ability to define atomicity and transaction behavior in the presence of concurrency, and an exception-like mechanism;
- It is a basis for reactive programming with a structured store such as a database. More specifically: it allows programming of reactive primitives which work on current and future states of the database, synchronization is naturally specified by the database state, and there are operators for concurrency, non-determinism, commit and isolation as provided in the form of transactions.

There are a number of related works which touch on different aspects of the following key properties:

- *Active Databases*
  The work here supports automatic trigerring of actions in response to selected changes in the database [16]. Thus there is some support for reactive programming within a database management system. However, active databases do not support reactive programming in general. This is partly because there is no systematic consideration of the global set of applications which are currently awaiting a response. In particular, concurrency constructs for inter-application coordination are not addressed. We say more about active databases below.
- *Process Calculi*
  The concern here is on concurrency and synchronization, and not reactivity to a common store, nor on concurrency control issues in databases. Typically the processes are distributed with little or no common store. Using the π-calculus [11] as a canonical example, we show in the appendix how this can be embedded.

- *Reactive Programming Languages*

  One class of reactive programming languages which deal with reactive programming are the synchronous languages such as Esterel [1], Lustre [4], etc. These languages are grounded in the concept of perfect synchronous concurrency where concurrent computations can be done in zero time. (These languages are further designed for deterministic reaction.) While suitable applications such as automatic control and hardware design, these languages are not suitable in a database context, partly because they are designed to react to signals only, and not a complicated store. Further, in order to be compatible with the perfect synchronous hypothesis of instantaneous reaction, programs in these languages have to be severely restricted. In contrast, we are here more concerned with general reactive programming which may have the more traditional non-deterministic concurrency aspects.

  The notion of synchronization using declarative conditions has been explored extensively in constraint programming [12]. In particular, the concurrent constraint programming (CCP) languages [14], use the notion of "ask" as the essential primitive synchronization mechanism. The ask constraint enables blocking until the relevant constraint is entailed by the store, and the unblocking is combined with a committed choice of alternatives. The "tell" constraint updates the store. Further, the updates are monotonic in the sense that true constraints always remain true. Our concern here involves non-monotonic stores. Further, CCP does not address database issues, and it addresses issues such as atomicity and transactions only in a limited way.

- *Coordination Languages*

  Coordination languages, such as blackboard-like languages and Linda [2], have some notion of a structured store and synchronization on the store. The blackboard tuple store is essentially a simple database. Synchronization is based upon certain tuple existing in the store. In the example of Linda, it also has data parallel features, eg. *eval*, to fill in tuple fields using concurrent processes. Thus, languages like Linda use a simple structured store with some synchronization and parallelism constructs. They do not address reactivity sufficiently generally, nor database issues such as atomicity and transactions.

Our framework combines various concerns in reactive programming, database, concurrent programming and concurrency control in a uniform formal setting. We present a basic calculus which also serves as a basis for various extensions. For example, an extension of basic concepts to include optimistic concurrency control is presented in the appendix.

In the end, our calculus provides for an open reactive programming environment where reactive database programs can be embedded within arbitrary concurrent agents (processes). They perform updates, with considerations of atomicity and transactional behavior, and synchronize with each other using the basic mechanisms available in the database system itself.

2 The Reactor Language

We consider here a language $\mathcal{R}$ for specifying *reactors*, a reactive program for interaction with a dynamically changing database. The syntax of $\mathcal{R}$ is given in the grammar below. The basic expressions c and u represent *functions* on the underlying database, and are defined simply as terms over a fixed signature of functors Σ and variables $\mathcal{V}$. The expression c is restricted to be a *test* on the database, and in general represents any database query without update. A test returns true or false. The notation u represents a general update. Reactors written in $\mathcal{R}$, are executed in much the same way as an SQL query, and may be embedded within an arbitrary agent. At any time, there are potentially many concurrent reactors executing or waiting for synchronization.

$$
\begin{array}{rcll}
r & ::= & \textsc{perform}\ u & \text{update} \\
 & | & c \Rightarrow_n u & \text{trigger immediately} \\
 & | & c \leadsto_n r & \text{trigger eventually} \\
 & | & r_1 \,\&\, r_2 & \text{interleave} \\
 & | & r_1 \mid r_2 & \text{early choice} \\
 & | & r_1 : r_2 & \text{late choice} \\
 & | & r_1 \,;\, r_2 & \text{sequencing} \\
 & | & [r] & \text{``atomic'' transaction} \\
 & | & \textsc{replicate}\ r\ \textsc{until}\ c & \text{uninterruptable replication} \\
 & | & \textsc{replicate}\ r\ \textsc{watch}\ c & \text{interruptable replication}
\end{array}
$$

Reactor expressions are subject to *substitutions*, mappings from variables into terms defined in the usual way. Now consider the intuitive semantics concerning the variables. We assume that the scope of a variable appearing in c in reactors of the form

$$c \Rightarrow_n u \quad \text{and} \quad c \leadsto_n r$$

is the entire expression. That is, a variable appearing in both c and u or r denotes the same variable. In contrast, occurrences of a variable appearing in both parts r_1 and r_2 of the composite expressions

$$r_1 \,\&\, r_2 \qquad r_1 \mid r_2 \qquad r_1 : r_2 \qquad r_1 \,;\, r_2$$

and nowhere else, concern two different variables, one for each part. That is, the scope of the variable which appears in r_1 is r_1 itself; similarly for r_2. Similarly, occurrences of a variable appearing in parts r and c of

$$\textsc{replicate}\ r\ \textsc{until}\ c \quad \text{and} \quad \textsc{replicate}\ r\ \textsc{watch}\ c$$

and nowhere else, concern two different variables.

Intuitively, the meaning of the reactor constructs are as follows. The straightforward constructs are: (i) update — performs some basic database update function; (ii) sequencing — reactor r_1 executes before r_2; and (iii) interleaving — runs both reactors concurrently in an interleaving fashion. The two choice constructs, early and late choice, are used to express non-deterministic alternatives. With early choice, the first reactor which can make progress is non-deterministically chosen to execute while the other alternative reactor is aborted without any effect. With late choice, the first reactor that terminates will stop the alternative reactor, but all effects of both reactors remain.

Synchronization on the state of the database is achieved by using the two trigger constructs. The trigger $c \Rightarrow_n u$ executes the update u when the condition c (which may be any database test) is true of the current state of the database when the reactor is executed, or when c is made true by some update. Further, the test c and update u are performed atomically in the sense that c is true when u is executed. Because of this, there is an effect of locking the database until u is done. For this reason, we do not permit a reactor such as $c \Rightarrow_n r$ where r is an arbitrary reactor. The trigger $c \leadsto_n r$, on the other hand, will execute the reactor r sometime after condition c is true. Thus c may not be true when r executes.

The subscript n in $c \leadsto_n r$ and $c \Rightarrow_n u$ is used to denote that n (not necessarily distinct) instantiations θ_1, ..., θ_n that make $c\theta_1$, ..., $c\theta_n$ all true are used to execute $r\theta_1$, ..., $r\theta_n$ (eventually) and $u\theta_1$, ..., $u\theta_n$ (immediately). As special cases, we also allow the constructs $c \leadsto r$ and $c \Rightarrow u$. These special-case constructs capture as many distinct instantiations as would make c true at the moment, that is, n need not be specified explicitly beforehand.

The atomic transaction construct, $[r]$, isolates reactor r from any updates and also isolates the database from transaction r until it has terminated. Intuitively, at the start of the transaction, r uses a copy of the database, updates made by r are not yet visible. At the end of the transaction, the updates made by r have to be reconciled with the state of the database and this is done by an operator $\dagger$ which needs to be defined appropriately.

The replication constructs, REPLICATE r UNTIL c and REPLICATE r WATCH c, run an indefinite number of reactors r concurrently which are terminated by condition c. The difference between uninterruptable and interruptable is whether c stops the replication of r after r completes or immediately which may stop r part-way through execution. Thus, WATCH behaves like an exception mechanism to interrupt r. Although replication runs copies of reactors concurrently, it can also express sequential iteration.

We end this section with a few examples. In what follows, we assume that given updates u and u', we can compose them into a compound update function $u \circ u'$. We use ! to denote the special "do-nothing" update function. We will use the unary operators $+$ and $-$, such as in $+u$ and $-u$, to denote the insertion and deletion of a tuple u from a relation. We will write relations using a predicate notation. Finally, to emphasize the scoping for variables, the first occurence of a variable $\mathbf{x}$ is denoted in bold, to indicate that it binds other occurences of x in its scope.

2.1 Examples

Inventory control Suppose we have a company which sells widgets, and a customer can order them in some quantity, denoted by the relation $ord(i, n)$ where i is the invoice. The company is headquartered at $\underline{\text{hq}}$. The relation $loc(w, n)$ describes current inventory n of a location w. Policy is that, given an order, one can either satisfy it with stock from $\underline{\text{hq}}$ if there is enough; alternatively, generate a request to ship widgets to $\underline{\text{hq}}$ other locations. The shipping request should not exceed max_w items per location.

Consider the following reactor. We assume that over time, the quantity of widgets in every location changes. We assume that the relation $\texttt{ship}(w,\ n)$ causes n items to be shipped from w to $\underline{\text{hq}}$ *eventually.*

$$
\begin{array}{l}
(\texttt{ord}(\mathbf{i}, \mathbf{n}) \wedge \texttt{loc}(\underline{\text{hq}}, \mathbf{m}) \wedge n < m \Rightarrow_1 -\texttt{ord}(i, n) + \texttt{ord}(i, 0)\ \ - \texttt{loc}(\underline{\text{hq}}, m) + \texttt{loc}(\underline{\text{hq}}, m - n)) \\
\mid \text{REPLICATE} \\
\quad \texttt{ord}(i, n) \wedge \texttt{loc}(w, m) \wedge w \neq \underline{\text{hq}} \wedge m > 0 \wedge x = min(m, n) \wedge x < max_w \Rightarrow_1 \\
\qquad -\texttt{ord}(i, n) + \texttt{ord}(i, x)\ \ - \texttt{loc}(w, m) + \texttt{loc}(w, m - x)\ \ + \texttt{ship}(w, x) \\
\text{UNTIL } \texttt{ord}(i, 0)
\end{array}
$$

This reactor uses early choice. Whenever there are enough items in $\underline{\text{hq}}$ to meet the order, and there is some stock in some warehouse, we non-deterministically choose between one of them. Note that the reactor waits until one of these two conditions hold, and commits to one of them, even if the other becomes true later. Consider now changing the choice construct above to the late choice construct. We in fact get a different behavior. It is now possible to account for some of the order using the shipping sub-reactor, and the rest elsewhere. One could further refine this example handle cancellation of shipping requests, having more preferences and strategies for dealing with an order, etc.

Locks It is also easy to implement control of concurrent access to resources, as shown in the following example. Let T be a resource that allows multiple simultaneous read access but exclusive write access. Access to it can be implemented using the following "design patterns". We assume the predicate $\texttt{reads}_T(n)$ that holds if and only if there are currently n active readers of T. We assume the predicate $\texttt{writes}_T(n)$ that holds if and only if there are currently n active writers of T. Then read access to T should be implemented according to the following "pattern":

$$
\begin{array}{l}
\texttt{writes}_T(0)\ \wedge\ \texttt{reads}_T(\mathbf{n})\ \Rightarrow_1\ -\texttt{reads}_T(n) + \texttt{reads}_T(n + 1)\ ; \\
\texttt{...read access to } T\texttt{...}\ ; \\
\texttt{reads}_T(\mathbf{m})\ \Rightarrow_1\ -\texttt{reads}_T(m) + \texttt{reads}_T(m - 1)
\end{array}
$$

Similarly, write access to T should be implemented according to the following pattern:

$$
\begin{array}{l}
\texttt{reads}_T(0)\ \wedge\ \texttt{writes}_T(0)\ \Rightarrow_1\ -\texttt{writes}_T(0) + \texttt{writes}_T(1)\ ; \\
\texttt{...write access to } T\texttt{...}\ ; \\
\text{PERFORM } - \texttt{writes}_T(1) + \texttt{writes}_T(0)
\end{array}
$$

Recursion A reactor system can perform recursive queries such as transitive closure of unordered graphs. Let $P(x, y)$ mean there is an edge from x to y. We want to create a relation $A(x, y)$ that holds if and only if there is a path from x to y. In the reactor below, we take the liberty of allowing a quantified expression as a database test.

$$
\begin{array}{l}
P(\mathbf{x}, \mathbf{y}) \ \Rightarrow \ +A(x, y) \ ; \\
\text{REPLICATE} \\
\quad A(\mathbf{x}, \mathbf{z}) \ \wedge \ A(z, \mathbf{y}) \ \wedge \ \neg A(x, y) \ \Rightarrow \ +A(x, y) \\
\text{UNTIL } \forall \mathbf{x}, \mathbf{z}, \mathbf{y}.\ A(x, z) \ \wedge \ A(z, y) \ \rightarrow \ A(x, y)
\end{array}
$$

Sequential Iteration Both the REPLICATE r UNTIL c and the REPLICATE r WATCH c constructs iterate r. Moreover, the different iterations of r are allowed to interleave each other. It is possible to obtain iterations that are strictly sequential and do not overlap each other.

It is possible to define variations of the classic repeat-until construct in our reactor language. Assume that $L(x)$ is a fresh predicate. Then the REPEAT $-$ WATCH construct can be defined as:

$$
\begin{array}{l}
\text{REPEAT } r \text{ WATCH } c \ = \\
\quad \text{PERFORM } + L(1) \ ; \\
\quad \text{REPLICATE} \\
\quad\quad L(\mathbf{x}) \ \Rightarrow_1 \ -L(x) \ ; \\
\quad\quad (r \ ; \ \text{PERFORM } + L(1)) \ : \ (c \ \Rightarrow_0 \ !) \ ; \\
\quad \text{WATCH } c \ ; \\
\quad \text{PERFORM } - L(1)
\end{array}
$$

Similarly for the REPEAT $-$ UNTIL case.

3 Operational Model

3.1 States and Deltas

A database is characterized by a set of *states*. At the user level, operations on a database are performed by tests and updates, as explained above. These operations are too high-level to be useful for the purpose of defining an operational semantics for reactors. Toward this goal, we now define a finer notion of *delta* operation on databases. Essentially, every reactor has access to a state Δ, and every execution step of a reaction is "annotated" by a delta δ. The definition of states and deltas, given below, are mutually recursive.

A state Δ comprises

- a "theory" FACTS(Δ)
- a finite set of "noted deltas" NOTED(Δ)
- a finite set of "pending deltas" PENDING(Δ)

The theory thus represents the usable or extensional meaning of the database. It can be formalized in many ways: as a relational database, a constraint database [9], a constraint store [12], a set of axioms, etc. All that we require is a notion of entailment and theory update. In what follows, we write $\textsc{Facts}(\Delta) \models c$ to denote that the test c is true of the database in state Δ.

The noted deltas are formal acknowledgement of the existence of a delta, and the pending deltas are a formal acknowledgement that a delta is "in process". Details follow below.

A delta δ has one of the following forms:

$\textsc{birth}\ r$
$\textsc{perform}\ u$
$\textsc{abort}\ r$
$i : \bullet\ c \leadsto_n r$
$i : c \leadsto_n^{\theta_1,\ldots,\theta_n} \bullet\ r$
$i : c \leadsto_n^{\theta_1,\ldots,\theta_n} r\ \bullet$
$i : \bullet\ c \Rightarrow_n u$
$i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} u\ \bullet$
$a : \bullet\ r, \Delta$
$a : r, \Delta \xrightarrow{\delta'} \bullet\ \Delta', r'$
$a : r, \Delta \xrightarrow{\delta'} \Delta'\ \bullet$

A delta that has one of the forms

$$i : \bullet\ c \leadsto_n r \qquad i : \bullet\ c \Rightarrow_n u \qquad a : \bullet\ r, \Delta$$

is called a *noted delta*. The i in a delta that has one of the forms

$$i : \bullet\ c \leadsto_n r \qquad i : c \leadsto_n^{\theta_1,\ldots,\theta_n} \bullet\ r \qquad i : c \leadsto_n^{\theta_1,\ldots,\theta_n} r\ \bullet$$
$$i : \bullet\ c \Rightarrow_n u \qquad i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} u\ \bullet$$

is called the *trigger identifier* of the delta. The a in a delta that has one of the forms

$$a : \bullet\ r, \Delta \qquad a : r, \Delta \xrightarrow{\delta'} \bullet\ \Delta', r' \qquad a : r, \Delta \xrightarrow{\delta'} \Delta'\ \bullet$$

is called the "transaction identifier" of the delta. In a delta that has one of these forms

$$i : \bullet\ c \leadsto_n r \qquad i : c \leadsto_n^{\theta_1,\ldots,\theta_n} \bullet\ r \qquad i : c \leadsto_n^{\theta_1,\ldots,\theta_n} r\ \bullet$$
$$i : \bullet\ c \Rightarrow_n u \qquad i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} u\ \bullet$$

it is assumed, where $FV(e)$ denotes the free variables in the expression e, that $FV(u) \subset FV(c)$ and $FV(r) \subset FV(c)$. Each trigger identifier or a transaction identifier is allowed to occur at most once in $\textsc{Noted}(\Delta) \cup \textsc{Pending}(\Delta)$. Only noted deltas are allowed in $\textsc{Noted}(\Delta)$ and only deltas of the form $i : c \leadsto_n^{\theta_1,\ldots,\theta_n} \bullet\ r$ are allowed in $\textsc{Pending}(\Delta)$.

We assume that there is an operation $\dagger$ for merging the theories of two states. As an example, if the theory of a state is a finite set of ground facts, $\dagger$ could be just the set union operation. We say more about this merge operation below.

We define $\textsc{Clean}(\Delta)$ to be a state Δ' derived from the state Δ by copying the latter's theory and setting the other components to empty; that is, $\textsc{Facts}(\Delta') = \textsc{Facts}(\Delta)$, $\textsc{Noted}(\Delta') = \{\}$, and $\textsc{Pending}(\Delta') = \{\}$. Also, we define $\textsc{abort}_{I,A}(\Delta)$ to be a state Δ' derived from the state Δ by aborting all the triggers identified in I and all the transactions identified in A; that is, $\textsc{Facts}(\Delta') = \textsc{Facts}(\Delta)$, $\textsc{Noted}(\Delta') = \{\delta \mid \delta \in \textsc{Noted}(\Delta)$, the trigger identifier of δ (if any) is not in I, the transaction identifier of δ (if any) is not in $A\}$, $\textsc{Pending}(\Delta') = \{\delta \mid \delta \in \textsc{Pending}(\Delta)$, the trigger identifier of δ (if any) is not in I, the transaction identifier of δ (if any) is not in $A\}$.

3.2 Operational Semantics

A reactor system is composed of a state Δ and a finite set of reactors R. The dynamic behavior of such a reactor system is defined by the transition relation $R, \Delta \longrightarrow_\delta \Delta', R'$, defined in this section. This reactor system transition relation is defined in terms of the reactor transition relation $r, \Delta \longrightarrow_\delta \Delta', r'$ and the reactor termination relation $r, \Delta \longrightarrow_\delta \Delta'$ of individual reactors r and the state transition relation $\Delta \longrightarrow_\delta \Delta'$.

Before proceeding, we add three new basic reactors to the language $\mathcal{R}$ defined above:

$\text{IDLE}_i \qquad \text{XACT}_a \qquad \text{ABORT } r$

These three constructs are not part of the language seen by a programmer. They are intermediate expressions produced by our operational semantics. The i in $\textsc{idle}_i$ is a trigger identifier returned by our operational semantics when a trigger is noted for later execution. The a in $\textsc{xact}_a$ is a transaction identifier returned by our operational semantics when an "atomic" transaction is noted for execution.

The basic version of the reactor system transition relation $R, \Delta \xrightarrow{\delta} \Delta', R'$ is given below. We use the notation $X + x$ to mean a set Y such that $Y = X \cup \{x\}$ and $x \notin X$. We use the notation $X - x$ to mean a set Y such that $X = Y + x$.

***** Birth *****

$$\frac{\Delta \xrightarrow{\delta} \Delta' \quad \delta \text{ is } \textsc{birth } r}{R, \Delta \xrightarrow{\delta} \Delta', R + r}$$

***** Death *****

$$\frac{r, \Delta \xrightarrow{\delta} \Delta'}{R + r, \Delta \xrightarrow{\delta} \Delta', R}$$

***** Progress *****

$$\frac{r, \Delta \xrightarrow{\delta} \Delta', r'}{R + r, \Delta \xrightarrow{\delta} \Delta', R + r'}$$

The "progress" transition rule given above is nondeterministic because a random reactor

r is picked from the reactor set and allowed to progress one step further. This rule can be substituted with other rules to reflect a different way for choosing which reactor to run next. For example, we can add a side condition to this rule to say that r should preferably contain a subexpression $\textsc{idle}_i$ such that the condition c of its corresponding noted trigger $i : \bullet\ c \leadsto_n\ r$ can be satisfied in the current state Δ.

Now we provide the rules for defining the reactor transition relation $r, \Delta \xrightarrow{\delta} \Delta', r'$ and the reactor termination relation $r, \Delta \xrightarrow{\delta} \Delta'$. Both of these relations have fairly complex side conditions, which we abstracted out in the state transition relation $\Delta \xrightarrow{\delta} \Delta'$ to be defined later.

***** Interleaving *****

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta', r_1'}{r_1\ \&\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1'\ \&\ r_2} \qquad \frac{r_2, \Delta \xrightarrow{\delta} \Delta', r_2'}{r_1\ \&\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1\ \&\ r_2'}$$

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta'}{r_1\ \&\ r_2, \Delta \xrightarrow{\delta} \Delta', r_2} \qquad \frac{r_2, \Delta \xrightarrow{\delta} \Delta'}{r_1\ \&\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1}$$

***** Sequencing *****

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta', r_1'}{r_1;\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1';\ r_2} \qquad \frac{r_1, \Delta \xrightarrow{\delta} \Delta'}{r_1;\ r_2, \Delta \xrightarrow{\delta} \Delta', r_2}$$

***** Triggering eventually *****

$$\frac{\begin{array}{c}\Delta \xrightarrow{\delta} \Delta' \\ \delta \text{ is } i : \bullet\ c \leadsto_n\ r \quad i \text{ is fresh}\end{array}}{c \leadsto_n\ r, \Delta \xrightarrow{\delta} \Delta', \textsc{idle}_i} \qquad \frac{\begin{array}{c}\Delta \xrightarrow{\delta} \Delta' \\ \delta \text{ is } i : c \leadsto_n^{\theta_1,\ldots,\theta_n}\ \bullet\ r\end{array}}{\textsc{idle}_i, \Delta \xrightarrow{\delta} \Delta', \textsc{idle}_i}$$

$$\frac{\begin{array}{c}\Delta \xrightarrow{\delta} \Delta' \\ \delta \text{ is } i : c \leadsto_n^{\theta_1,\ldots,\theta_n}\ r\ \bullet\end{array}}{\textsc{idle}_i, \Delta \xrightarrow{\delta} \Delta', r\theta_1\ \&\ \cdots\ \&\ r\theta_n}$$

***** Triggering immediately *****

$$\frac{\begin{array}{c}\Delta \xrightarrow{\delta} \Delta' \\ \delta \text{ is } i : \bullet\ c \Rightarrow_n\ u \quad i \text{ is fresh}\end{array}}{c \Rightarrow_n\ u, \Delta \xrightarrow{\delta} \Delta', \textsc{idle}_i} \qquad \frac{\begin{array}{c}\Delta \xrightarrow{\delta} \Delta' \\ \delta \text{ is } i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n}\ u\ \bullet\end{array}}{\textsc{idle}_i, \Delta \xrightarrow{\delta} \Delta'}$$

***** Update and abortion *****

$$\frac{\begin{array}{c}\Delta \xrightarrow{\delta} \Delta' \\ \delta \text{ is } \textsc{perform}\ u\end{array}}{\textsc{perform}\ u, \Delta \xrightarrow{\delta} \Delta'} \qquad \frac{\begin{array}{c}\Delta \xrightarrow{\delta} \Delta' \\ \delta \text{ is } \textsc{abort}\ r\end{array}}{\textsc{abort}\ r, \Delta \xrightarrow{\delta} \Delta'}$$

***** Delayed choice *****

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta', r_1'}{r_1 \ :\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1' \ :\ r_2} \qquad \frac{r_2, \Delta \xrightarrow{\delta} \Delta', r_2'}{r_1 \ :\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1 \ :\ r_2'}$$

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta'}{r_1 \ :\ r_2, \Delta \xrightarrow{\delta} \Delta', \text{ABORT}\ r_2} \qquad \frac{r_2, \Delta \xrightarrow{\delta} \Delta'}{r_1 \ :\ r_2, \Delta \xrightarrow{\delta} \Delta', \text{ABORT}\ r_1}$$

***** Early choice *****

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta', r_1' \quad \delta \text{ is a noted delta}}{r_1 |\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1' |\ r_2} \qquad \frac{r_1, \Delta \xrightarrow{\delta} \Delta', r_1' \quad \delta \text{ is not a noted delta}}{r_1 |\ r_2, \Delta \xrightarrow{\delta} \Delta', \text{ABORT}\ r_2\ ;\ r_1'}$$

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta'}{r_1 |\ r_2, \Delta \xrightarrow{\delta} \Delta', \text{ABORT}\ r_2}$$

$$\frac{r_2, \Delta \xrightarrow{\delta} \Delta', r_2' \quad \delta \text{ is a noted delta}}{r_1 |\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1 |\ r_2'} \qquad \frac{r_2, \Delta \xrightarrow{\delta} \Delta', r_2' \quad \delta \text{ is not a noted delta}}{r_1 |\ r_2, \Delta \xrightarrow{\delta} \Delta', \text{ABORT}\ r_1\ ;\ r_2'}$$

$$\frac{r_2, \Delta \xrightarrow{\delta} \Delta'}{r_1 |\ r_2, \Delta \xrightarrow{\delta} \Delta', \text{ABORT}\ r_1}$$

***** Uninterruptable replication *****

$$\frac{(r\ \&\ \text{REPLICATE}\ r\ \text{UNTIL}\ c)\ |\ (c \Rightarrow_0 !), \Delta \xrightarrow{\delta} \Delta', r'}{\text{REPLICATE}\ r\ \text{UNTIL}\ c, \Delta \xrightarrow{\delta} \Delta', r'}$$

$$\frac{(r\ \&\ \text{REPLICATE}\ r\ \text{UNTIL}\ c)\ |\ (c \Rightarrow_0 !), \Delta \xrightarrow{\delta} \Delta'}{\text{REPLICATE}\ r\ \text{UNTIL}\ c, \Delta \xrightarrow{\delta} \Delta'}$$

***** Interruptable replication *****

$$\frac{(r\ \&\ \text{REPLICATE}\ r\ \text{WATCH}\ c)\ :\ (c \Rightarrow_0 !), \Delta \xrightarrow{\delta} \Delta', r'}{\text{REPLICATE}\ r\ \text{WATCH}\ c, \Delta \xrightarrow{\delta} \Delta', r'}$$

$$\frac{(r\ \&\ \text{REPLICATE}\ r\ \text{WATCH}\ c)\ :\ (c \Rightarrow_0 !), \Delta \xrightarrow{\delta} \Delta'}{\text{REPLICATE}\ r\ \text{WATCH}\ c, \Delta \xrightarrow{\delta} \Delta'}$$

***** Transactions *****

$$\frac{\Delta \xrightarrow{\delta} \Delta' \quad \delta \text{ is } a : \bullet\ r, \Delta \quad a \text{ is fresh}}{[r], \Delta \xrightarrow{\delta} \Delta', \text{XACT}_a} \qquad \frac{\Delta \xrightarrow{\delta} \Delta' \quad \delta \text{ is } a : r, \Delta_o \xrightarrow{\delta'} \bullet\ \Delta_n, r' \quad r, \Delta_o \xrightarrow{\delta'} \Delta_n, r'}{\text{XACT}_a, \Delta \xrightarrow{\delta} \Delta', \text{XACT}_a} \qquad \frac{\Delta \xrightarrow{\delta} \Delta' \quad \delta \text{ is } a : r, \Delta_o \xrightarrow{\delta'} \Delta_n\ \bullet \quad r, \Delta_o \xrightarrow{\delta'} \Delta_n}{\text{XACT}_a, \Delta \xrightarrow{\delta} \Delta'}$$

Here, we briefly explain some important points on the operational semantics. The eventual trigger, $c \ \leadsto_n \ r$, can be thought of as a two step process: first determine that

$c\theta_1, \ldots, c\theta_n$ is true, and schedule the reactor r for execution. Eventually, r is executed in the sense that n copies $r\theta_1, \ldots, r\theta_n$ are run concurrently. The immediate trigger, $c \Rightarrow_n u$, on the other hand, checks that $c\theta_1, \ldots, c\theta_n$ is true and then performs the updates $u\theta_1 \circ \cdots \circ u\theta_n$ atomically. In the early and late choice constructs, when there is committal, it is necessary to stop execution of reactors in the alternative choice. If the stopped reactors have already scheduled various triggers and transactions, then a cleanup is carried out by using $\textsc{abort}_{I,A}(\Delta)$. Replication is defined by using the choice constructs. Uninterruptable replication, $\textsc{replicate}\ r\ \textsc{until}\ c$, is defined using early choice. This is because the test c is ignored when the reactor r makes transitions. Conversely, interruptable replication is defined using late choice. This is to make the test c continuously active. Finally consider transactions. A reactor involved inside a transaction uses its own copy of the database. In the transition rules, this state is kept within the transaction δ.

As mentioned before, some complex side conditions for the above reactor transition and reactor termination relations are abstracted into a state transition relation $\Delta \xrightarrow{\delta} \Delta'$. We provide this state transition relation below.

$$\frac{\delta \text{ is } \textsc{birth}\ r}{\Delta \xrightarrow{\delta} \Delta} \qquad \frac{\begin{array}{c}\delta \text{ is } \textsc{perform}\ u \\ \textsc{Facts}(\Delta') = u(\textsc{Facts}(\Delta)) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\end{array}}{\Delta \xrightarrow{\delta} \Delta'} \qquad \frac{\begin{array}{c}\delta = \textsc{abort}\ r \\ I = \{i \mid \textsc{idle}_i \text{ appears in } r\} \\ A = \{a \mid \textsc{xact}_a \text{ appears in } r\}\end{array}}{\Delta \xrightarrow{\delta} \textsc{abort}_{I,A}(\Delta)}$$

$$\frac{\begin{array}{c}\delta = i : \bullet\ c \rightsquigarrow_n r \\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) + \delta \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\end{array}}{\Delta \xrightarrow{\delta} \Delta'} \qquad \frac{\begin{array}{c}\delta = i : c \rightsquigarrow_n^{\theta_1,\ldots,\theta_n} \bullet\ r \\ \textsc{Facts}(\Delta) \models c\theta_j \text{ for all } 1 \le j \le n \\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) - (i : \bullet\ c \rightsquigarrow_n r) \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta) + \delta\end{array}}{\Delta \xrightarrow{\delta} \Delta'}$$

$$\frac{\begin{array}{c}\delta = i : c \rightsquigarrow_n^{\theta_1,\ldots,\theta_n} r\ \bullet \\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta) - i : c \rightsquigarrow_n^{\theta_1,\ldots,\theta_n} \bullet\ r\end{array}}{\Delta \xrightarrow{\delta} \Delta'}$$

$$\frac{\begin{array}{c}\delta = i : \bullet\ c \Rightarrow_n u \\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) + \delta \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\end{array}}{\Delta \xrightarrow{\delta} \Delta'} \qquad \frac{\begin{array}{c}\delta = i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} u\ \bullet \\ \textsc{Facts}(\Delta) \models c\theta_j \text{ for all } 1 \le j \le n \\ \textsc{Facts}(\Delta') = (u\theta_1 \circ \cdots \circ u\theta_n)(\textsc{Facts}(\Delta)) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) - (i : \bullet\ c \Rightarrow_n u) \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\end{array}}{\Delta \xrightarrow{\delta} \Delta'}$$

$$\frac{\begin{array}{c}\delta = a : \bullet\ r, \textsc{Clean}(\Delta) \\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta)\ + \delta \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\end{array}}{\Delta \xrightarrow{\delta} \Delta'}$$

$$\frac{\begin{array}{c}\delta = a : r, \Delta_o \xrightarrow{\delta'} \bullet\ \Delta_n, r' \\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta) \\ \textsc{Noted}(\Delta') = (\textsc{Noted}(\Delta) - (a : \bullet\ r, \Delta_o)) + (a : \bullet\ r', \Delta_n) \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\end{array}}{\Delta \xrightarrow{\delta} \Delta'}$$

$$\frac{\begin{array}{c}\delta = a : r, \Delta_o \xrightarrow{\delta'} \Delta_n\ \bullet \\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta)\ \dagger\ \textsc{Facts}(\Delta_n) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) - a : \bullet\ r, \Delta_o \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\end{array}}{\Delta \xrightarrow{\delta} \Delta'}$$

3.3 Transactions

Each transaction $[r]$ executes within its own private state, and is *isolated* from the rest of the reactor system. The effect of such a transaction is then committed by merging its termination state with the current state of the entire reactor system, using the $\dagger$ operator. In conventional databases, transactions enjoy the ACID property, meaning that they enjoy *Atomicity*, Consistency, Isolation and Durability. Isolation in the conventional database context means that transactions can be considered without worrying about concurrency, that is, isolation equals serializability. In our calculus, we use isolation only to mean that while $[r]$ is executing, updates to the database are invisible to r and vice versa. Later, we will consider a modification to handle serializable transactions. Our operational semantics thus lets a transaction progress in a step-wise interleaved manner with the rest of the system without interference, avoiding the need to hold back other transactions. This is analogous to atomic transactions in an "optimistic" database system[15].

The treatment of transactions in our framework also differs from database in that because we have here a reactive programming framework with concurrency control. Some control can be exercised over issues like deadlock, aborting of transactions and recovery at the programming level rather than at the internal database level which is mostly invisible and beyond direct control of an application program.

Previously we left the $\dagger$ operator which reconciles the final state of the transaction with the rest of the system. We now examine $\dagger$ and $[r]$ and consider possible choices and modifications to the semantics. We do so in a database management system-like context. Without loss of generality, let us assume that for any state Δ, its associated theory $\textsc{Facts}(\Delta)$ is a finite set of atomic tuples, like a normal relational database. Similarly, we assume that the condition c in

$c \rightsquigarrow_n r \qquad c \Rightarrow_n u \qquad$ REPLICATE r UNTIL $c \qquad$ REPLICATE r WATCH c

is restricted to the form equivalent to some first-order formula QM, where Q is a quantifier prefix and M is a quantifier free matrix.

Union Merge: Earth9 and Lotus Notes Transactions

Earlier, one example of † was the set union operation. This sounds problematic since two transactions can take their own snapshots of the state, then independently make conflicting changes based on their isolated view of the state. Before we address this using traditional database techniques, we wish to point out that such a simple semantics for $[r]$ is not unreasonable for some reactive applications. In fact, it is precisely that of Lotus Notes, Earth9, and other simple transaction systems popular on the Internet.

We now turn to an example using the calendar system of Earth9 (`http://www.earth9.com`). Assume Wong and his secretary (Phyliss) use Earth9 to manage Wong's daily schedule. They may both choose at the same time to schedule a meeting, say with $[r_{wong}]$ and $[r_{phyllis}]$, which happens to be on the same free slot. The Earth9 calendar system, like other free calendar systems on the web, simply accepts both appointments even though there is a clash; ie. Earth9 is using the simple union operation as its † operator. However, Earth9 turns out to be well-designed. It will then return a confirmation page which lists all the scheduled meetings. So the conflict can be noticed and can be separately resolved. This is also the strategy adopted by the widely used Lotus Notes and other less well known multi-user collaboration systems. So the simple semantics assigned to the $[r]$ construct and the † operator is a realistic and reasonable choice.

It is worth pointing out, that a user-definable † operation is also potentially useful. A simple one is where treatment of different updates on different relations is specified, e.g. ignore deletions from the transaction on a particular relation.

Serializable Transactions

There are of course more demanding semantics from typical relational database management systems. Transactions in such database system can be executed concurrently with their operations interleaved. However, regardless of the interleaving, the scheduling of these operations must satisfy the serializability property from ACID. *Serializability* means that also the end results of interleaving these operations should be equivalent to some serial non-interleaved execution of these transactions [13]. In the classical setting, serializability of two interleaved transactions is violated when one transaction updates an item and the other transaction updates or reads that same item.

There are two basic approaches toward serializability: locking vs non-locking. A representative locking approach is two-phase locking [3]. Before reading or updating any item, a transaction must acquire a lock on that item and must promise to release this lock some time before it commits. Moreover, once a transaction releases any lock, it is not allowed to acquire any new lock. Using this approach, the two types of conflicts mentioned above can never arise. However, two transactions can be dead locked. When dead lock occurs, one of the transactions must be killed and its effects undone so that the other transaction can proceed. Later, the killed transaction must be restarted from scratch.

A representative non-locking approach is optimistic concurrency control [10]. A transaction gets a "slave" copy of the "master" database and updates its own copy. The transaction keeps a trace of the records it has read or updated. When the transaction attempts to commit, it first checks that all the records it has read or updated remain unchanged in the master database. If this check is satisfied, it commits its updates on its slave copy to the master database. If this check is not satisfied, then one of the two type of conflicts must have occurred; the transaction is aborted and later restarted with a new slave copy.

An optimistic concurrency control method for serializability is most natural in our setting. Let r be a reactor containing $[r']$ as a subexpression. We can then treat r as the "main" transaction operating on the "master" database and $[r']$ as a "side" transaction operating on a "slave" copy of the master database. However, we need to make some changes to operational semantics to incorporate the detection of conflicts and the restarting of aborted transactions. Due to lack of space, the details are relegated to the appendix. Other alternatives, such as two phase locking and transaction recovery between the free-for-all style of Lotus Notes and Earth9 also fit within our framework.

4 Active Databases

An active relational database system allows the definition of certain trigger rules in the ECA form[16]: ON *event* IF *condition* THEN *action*. The events are typically insert, delete, modify. The condition is some database test which can access both old and new values. The action can be any database query. There are some comparisons to make between a reactor system and an active relational database system.

ECA rules are defined to trigger on database events, and as such, are direct mechanisms addresses system specific characteristics within the database system. Our triggers, on the other hand, depend on the declarative or extensional semantics of the database. Typically ECA rules are used to maintain integrity constraints or repair such constraints. Our reactor system, on the other hand, is more concerned with an expressive and programmable framework, and focuses on a number of key mechanisms: immediate and delayed triggers, concurrency control, committed choice and transactions.

It is possible to implement our triggers using an active relational database system as the underlying database. Let c be a Boolean combination and u be simple deletions and insertions on tables. Then a reactor such as $c \leadsto$ PERFORM u can be implemented as a series of trigger rules like this: For each table mention in c and for each possible kind of update event e on this table, add the rule ON e IF c THEN u to the active relational database system. Hence a simple reactor can explode into a large number of trigger rules. Note that if the reactor was $c \leadsto_n$ PERFORM u, it would not be so easy to implement using

ECA rules, as these rules cannot capture the notion of choosing just n instantiations.

In commercial relational databases, some restrictions are imposed on ECA trigger rules [16]: The action in an ECA trigger rule is a combination of simple deletions from and insertions into relational tables; the condition in an ECA trigger rule is a Boolean combination; and recursive firing of ECA trigger rules is disallowed. Under such a setting, we claim that

Proposition 4.1 *A reactor system is strictly more expressive than typical ECA trigger rules.*

Proof Sketch. We first demonstrate inclusion. Let e be an event mentioned in the ECA rules. Let ON e IF c_i THEN u_i, for $1 \leq i \leq n$, be all ECA trigger rules whose triggerring event is e. In the corresponding reactor system, we assume there is a predicate $\mathtt{event}(e,t)$ that holds if e occurs at time t and has not been handled yet. Let $handle_i$ be the expression $(\mathtt{event}(e,\mathbf{t}) \wedge c_1 \Rightarrow u_1) : (\mathtt{event}(e,\mathbf{t}) \wedge \neg c_1 \Rightarrow !)$. Then these ECA triggers on event e can be implemented in our reactor system as (REPEAT $handle_1$ & $\cdots$ & $handle_n$ UNTIL $false$) ; PERFORM $-\,\mathtt{event}(e,t)$.

For strictness, we note the followings. These restricted ECA trigger rules cannot express recursive functions such as transitive closure on unordered graphs and they cannot capture the notion of just n instantiations. The first inability can be easily proved from the bounded degree property of SQL queries[8]. The second inability follows from the genericity of SQL queries on unordered graphs. On the other hand, our reactor system posseses both capabilities. $\square$

Finally, we remark that in relational database systems there is usually an ad-hoc procedural extension to the base query languages. This can be used within ECA rules to do programming. This provides for rather low level and system dependent programming.

5 Conclusion

We presented a calculus for the specification of concurrent processes which interact with a changing database. One design considerations is that synchronization and communication is obtained from the native database operations, and this can be modelled declaratively. Specifically, processes block on database tests which are defined using some notion of theory entailment, they update the database using native database operations, and there is a variety of constructs for dealing with concurrency issues such as atomicity and transactions. The calculus presented dealt with core issues, and it can be extended easily to accomodate more complicated database characteristics.

A prototype implementation is under way. At the heart of this system is a constraint logic program [5] representing a constraint database. The current implementation uses our own CLP($\mathcal{R}$) system [6], and a superstructure whose main purpose is the management of a collection of reactors. This system is an instance of a larger effort, Open Constraint Programming [7], which discusses a general architecture matching the calculus presented here. The system in particular provides knowledge representation of shared data using rules and constraints. Current applications targetted include workflow and resource allocation problems.

References

[1] G. Béry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming.* 19(2):87–152, 1992.

[2] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM.* 32(4):444–458, 1989.

[3] K.P. Eswaran, J,N, Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM.* 19(11):624–633, 1976.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE.* 79(9), 1991.

[5] J. Jaffar and M. Maher. Constraint Logic Programming: a Survey, *Journal of Logic Programming*, Special 10^{th} Anniversary Issue, Vols 19/20, May/July 1994.

[6] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. The CLP($\mathcal{R}$) Language and System, *ACM Transactions on Programming Languages and Systems*, 14(3), July 1992.

[7] J. Jaffar and R.H.C. Yap. Open Constraint Programming, Invited Paper, *4th Intl. Conf on Principles and Practice of Constraint Programming*, Oct 1998.

[8] L. Libkin and L. Wong. Query languages for bags and aggregate functions. *Journal of Computer and System Sciences.* 55(2):241–272, 1997.

[9] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences.* 51(1):26–52, 1995.

[10] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems.* 6(2):213–226, 1981.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I/II. *Information and Computation.* 100(1):1-77, 1992.

[12] K. Marriott and P.J. Stuckey, *Programming with Constraints.* The MIT Press, 1998.

[13] R. Ramakrishnan. *Database Management Systems.* McGraw-Hill, 1997.

[14] V. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[15] G. Schlageter. Optimistic methods for concurrency control in distributed systems. In *Proceedings of 7th International Conference on Very Large Data Bases.* 125–130, 1981.

[16] J. Widom, S. Ceri, and U. Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1995.

Appendix

Optimistic Concurrency Control

We consider an optimistic concurrency control method for serializability in our setting. Let r be a reactor containing $[r']$ as a subexpression. We treat r as the "main" transaction operating on the "master" database and $[r']$ as a "side" transaction operating on a "slave" copy of the master database. However, we need to make some changes to operational semantics to incorporate the detection of conflicts and the restarting of aborted transactions.

A state Δ now includes a "log", $\textsc{Log}(\Delta)$, which is a finite set of record of the form $r(u,t)$ or the form $w(u,t)$. The former means that a tuple u was read at time t. The latter means that a tuple u was updated at time t. Let M be the log of a side transaction and N be the log of the main transaction. M is compatible with N if the following two conditions hold. (i) For every $r(u,t) \in M$ and $w(u,t') \in N$, it is the case that $t > t'$. The violation of this condition implies the occurrence of a read-write conflict. (ii) For every $w(u,t) \in M$ and $w(u,t') \in N$, it is the case that $t > t'$. The violation of this condition implies the occurrence of a write-write conflict. Note that M and N can be compatible if $w(u,t) \in M$, $r(u,t') \in N$, and $t > t'$. This asymmetry arises from the distinction between side transactions (which run in isolation) and main transactions (which see everything).

We need to put more information into our syntax to facilitate restart of side transactions. This is accomplished by changing $\textsc{xact}_a$ to $\textsc{xact}_{a,r}$, where a is a transaction identifier as before and r is the original side transaction.

We need to add one more trigger to indicate that a transaction fails due to the presence of conflicts. This new trigger has the form $a : r, \Delta \xrightarrow{\delta'} \Delta'$ $\textsc{fail}$.

We need to add to the definition of $\textsc{abort}_{I,A}(\Delta)$ so that it produces a state Δ' that also satisfies $\textsc{Log}(\Delta') = \textsc{Log}(\Delta)$. Similarly, we need to add to the definition of $\textsc{Clean}(\Delta)$ so that it produces a state Δ' that also satisfies $\textsc{Log}(\Delta') = \{\}$.

We need to add a notion of time into all our transition relations that define the operational semantics. The way to do this is straightforward. We use some examples to illustrate. In these examples, t is the time before the transition and t' is the time after the transition. For this paper, we assume discrete time; so $t' = t + 1$ always.

- For the "progress" transition rule of the reactor system transition relation, we change it to

$$\frac{t, r, \Delta \xrightarrow{\delta} \Delta', r', t'}{t, R + r, \Delta \xrightarrow{\delta} \Delta', R + r', t'}$$

- For the "transaction" rules of the reactor transition relation, we change them to

$$\frac{t, \Delta \xrightarrow{\delta} \Delta', t' \qquad \delta \text{ is } a : \bullet\, r, \Delta \quad a \text{ is fresh}}{t, [r], \Delta \xrightarrow{\delta} \Delta', \textsc{xact}_{a,r}, t'}
\qquad
\frac{t, \Delta \xrightarrow{\delta} \Delta', t' \qquad \delta \text{ is } a : r, \Delta_o \xrightarrow{\delta'} \bullet\, \Delta_n, r' \qquad t, r, \Delta_o \xrightarrow{\delta'} \Delta_n, r', t'}{t, \textsc{xact}_{a,r_o}, \Delta \xrightarrow{\delta} \Delta', \textsc{xact}_{a,r_o}, t'}$$

$$\frac{t, \Delta \xrightarrow{\delta} \Delta', t' \qquad \delta \text{ is } a : r, \Delta_o \xrightarrow{\delta'} \Delta_n \bullet \qquad t, r, \Delta_o \xrightarrow{\delta'} \Delta_n, t'}{t, \textsc{xact}_{a,r_o}, \Delta \xrightarrow{\delta} \Delta', t'}
\qquad
\frac{t, \Delta \xrightarrow{\delta} \Delta', t' \qquad \delta \text{ is } a : r, \Delta_o \xrightarrow{\delta'} \Delta_n \text{ FAIL} \qquad t, r, \Delta_o \xrightarrow{\delta'} \Delta_n, t'}{t, \textsc{xact}_{a,r_o}, \Delta \xrightarrow{\delta} \Delta', r_o, t'}$$

The last "transaction" rule above is added to handle the restart of a transaction that fails.

Lastly, we need to add logs as well as time into the state transition relations. We give the more important cases for illustration. The remaining cases are straightforward.

$$\frac{\begin{array}{c}\delta \text{ is } \textsc{perform } u \\ \textsc{Facts}(\Delta') = u(\textsc{Facts}(\Delta)) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta) \\ \textsc{Log}(\Delta') = \textsc{Log}(\Delta) \cup U \\ U = \{w(u', t) \mid u' \text{ is an item updated by } u\}\end{array}}{t, \Delta \xrightarrow{\delta} \Delta', t+1}
\qquad
\frac{\begin{array}{c}\delta = \textsc{abort } r \\ I = \{i \mid \textsc{idle}_i \text{ appears in } r\} \\ A = \{a \mid \textsc{xact}_{a,r_o} \text{ appears in } r\}\end{array}}{t, \Delta \xrightarrow{\delta} \textsc{abort}_{I,A}(\Delta), t+1}$$

$$\frac{\begin{array}{c}\delta = i : c \leadsto_n^{\theta_1,\ldots,\theta_n} \bullet\, r \\ \textsc{Facts}(\Delta) \models c\theta_j \text{ for all } 1 \le j \le n \\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) - (i : \bullet\, c \leadsto_n r) \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta) + \delta \\ \textsc{Log}(\Delta') = \textsc{Log}(\Delta) \cup R \\ R = \{r(u', t) \mid u' \text{ is an item read during the test } \textsc{Facts}(\Delta) \models c\theta_j \text{ for all } 1 \le j \le n\}\end{array}}{t, \Delta \xrightarrow{\delta} \Delta', t+1}$$

$$\frac{\begin{array}{c}\delta = i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} u \bullet \\ \textsc{Facts}(\Delta) \models c\theta_j \text{ for all } 1 \le j \le n \\ \textsc{Facts}(\Delta') = (u\theta_1 \circ \cdots \circ u\theta_n)(\textsc{Facts}(\Delta)) \\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) - (i : \bullet\, c \Rightarrow_n u) \\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta) \\ \textsc{Log}(\Delta') = \textsc{Log}(\Delta) \cup U \cup R \\ U = \{w(u', t) \mid u' \text{ is an item updated by } u\theta_j \text{ for any } 1 \le j \le n\} \\ R = \{r(u', t) \mid u' \text{ is an item read during the test } \textsc{Facts}(\Delta) \models c\theta_j \text{ for all } 1 \le j \le n\}\end{array}}{t, \Delta \xrightarrow{\delta} \Delta', t+1}$$

$$\frac{\begin{array}{c}\delta = a : \bullet\ r, \textsc{Clean}(\Delta)\\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta)\\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta)\ + \delta\\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\\ \textsc{Log}(\Delta') = \textsc{Log}(\Delta)\end{array}}{t, \Delta \xrightarrow{\delta} \Delta', t+1}$$

$$\frac{\begin{array}{c}\delta = a : r, \Delta_o \xrightarrow{\delta'} \bullet\ \Delta_n, r'\\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta)\\ \textsc{Noted}(\Delta') = (\textsc{Noted}(\Delta) - (a : \bullet\ r, \Delta_o)) + (a : \bullet\ r', \Delta_n)\\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\\ \textsc{Log}(\Delta') = \textsc{Log}(\Delta)\end{array}}{t, \Delta \xrightarrow{\delta} \Delta', t+1}$$

$$\frac{\begin{array}{c}\delta = a : r, \Delta_o \xrightarrow{\delta'} \Delta_n\ \bullet\\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta) \dagger \textsc{Facts}(\Delta_n)\\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) - a : \bullet\ r, \Delta_o\\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\\ \textsc{Log}(\Delta') = \textsc{Log}(\Delta)\\ \textsc{Log}(\Delta_n) \text{ is compatible with } \textsc{Log}(\Delta)\end{array}}{t, \Delta \xrightarrow{\delta} \Delta', t+1} \qquad \frac{\begin{array}{c}\delta = a : r, \Delta_o \xrightarrow{\delta'} \Delta_n\ \textsc{fail}\\ \textsc{Facts}(\Delta') = \textsc{Facts}(\Delta)\\ \textsc{Noted}(\Delta') = \textsc{Noted}(\Delta) - a : \bullet\ r, \Delta_o\\ \textsc{Pending}(\Delta') = \textsc{Pending}(\Delta)\\ \textsc{Log}(\Delta') = \textsc{Log}(\Delta)\\ \textsc{Log}(\Delta_n) \text{ is not compatible with } \textsc{Log}(\Delta)\end{array}}{t, \Delta \xrightarrow{\delta} \Delta', t+1}$$

Thus we see that with very simple changes to the operational semantics, we can introduce the optimistic concurrency control with full transaction recovery of traditional database management systems into our reactor system.

Embedding the π-Calculus

We compare reactors r to expressions P in π-calculus [11]. Let us first recall π-calculus. The syntax of π-calculus is

$$\pi ::= x(\mathbf{y}) \mid \hat{x}\langle y\rangle \mid \tau$$
$$P ::= \pi_1.P_1\ +\ \cdots\ +\ \pi_n.P_n \mid P_1 \mid P_2 \mid \mathit{new}\ \mathbf{x}\ P \mid !\ P \mid 0$$

The semantics of π-calculus is defined in terms of a set of reaction rules modulo a number of structural congruences. The structural congruences are alpha conversion; $+$ is associative and commutative; $\mid$ is associative, commutative, and has 0 as identity element; $\mathit{new}\ \mathbf{x}\ (P \mid Q) \equiv\ P\ \mathit{new}\ \mathbf{x}\ Q$, if x not free in P; $\mathit{new}\ \mathbf{x}\ 0 \equiv\ 0$; and $!\ P \equiv\ P \mid !\ P$. The reaction rules are $\tau.P\ +\ M \leadsto\ P$; $(x(\mathbf{y}).P\ +\ M) \mid (\hat{x}\langle z\rangle.Q\ +\ N) \leadsto\ \{z/y\}P \mid Q$; $P \mid Q\ \leadsto\ P' \mid Q$, if $P\ \leadsto\ P'$; $\mathit{new}\ \mathbf{x}\ P\ \leadsto\ \mathit{new}\ \mathbf{x}\ P'$, if $P\ \leadsto\ P'$; and $Q\ \leadsto Q'$, if $P\ \leadsto\ P'$ and $Q\ \equiv P$ and $Q'\ \equiv P'$. In the proofs of our theorems, we assume a slightly modified set of structural congruences and reaction rules. In particular, (a) we remove the two structural congruences involving the $\mathit{new}\ \mathbf{x}\ P$ construct; and (b) we replace the reaction rule: $\mathit{new}\ \mathbf{x}\ P\ \leadsto\ \mathit{new}\ \mathbf{x}\ P'$, if $P\ \leadsto\ P'$ by the new reaction rule: $\mathit{new}\ \mathbf{x}\ P\ \leadsto \{z/x\}P$, if z is fresh. The effects of the modified system is equivalent to the original one modulo some obvious translations. We made these modifications to simplify our work in the present context.

We now attempt a mapping of π-calculus into our reactor language. We assume that there is a special predicate in our reactor language $C(x, y, s)$ that holds iff there is a "message" y along "channel" x with "timestamp" s. We assume that there is a special predicate in our reactor language $C'(x, y, s)$ that holds iff there is a message y along channel x with timestamp s that has just been read. We assume that there is a special predicate in our reactor language $C''(x, y, s, t)$ that holds iff there is a message y along channel x with timestamp s that has just been read at timestamp t. We assume there is a special predicate in our reactor language $new(x)$ that holds iff x is "fresh"; that is, it is a counter for new names. We assume that there is a special flag in our reactor N for controlling access to $new(x)$. We assume there is a special predicate in our reactor language $timestamp(s)$ that holds iff s is "fresh"; that is, it is a counter for new timestamps. We assume that there is a special flag in our reactor S for controlling access to $timestamp(s)$. Also, we ignore $\tau.P$, since this corresponds to "hidden" action. Let $\overline{P}$ be a reactor derived from the π-calculus expression P as follows:

- $\overline{0} = \textsc{perform}\ !$
- $\overline{!\,P} = \textsc{replicate}\ \overline{P}\ \textsc{watch}\ \mathit{false}$
- $\overline{P_1 \mid P_2} = \overline{P_1}\ \&\ \overline{P_2}$
- $\overline{\mathit{new}\ \mathbf{x}\ P} = (\neg N \ \Rightarrow_1\ +N)\ ;\ (\mathit{new}(\mathbf{x})\ \rightsquigarrow_1\ (\textsc{perform}\ -\mathit{new}(x) + \mathit{new}(x+1) - N\ ;\ \overline{P}))$

  The first part $\neg N \ \Rightarrow_1\ +N$ is a trick for guaranteeing exclusive access to $new(\mathbf{x})$, in a similar spirit to example of readers and writers. We also take liberty here with variable scoping, and abuse the syntax $\mathbf{x}$ to now mean that the scope of x is from the first $\mathbf{x}$ to the next occurrence of $\mathbf{x}$. This shortcut, while formally not covered above, is taken just for simplicity.

- $\overline{\pi_1.P_1\ +\ \cdots\ +\ \pi_n.P_n} = (\neg S\ \Rightarrow_1\ +S)\ ;\ (\mathit{timestamp}(\mathbf{s})\ \rightsquigarrow_1\ (\textsc{perform}\ -\mathit{timestamp}(s) + \mathit{timestamp}(s+1) - S\ ;\ (\overline{\pi_1.P_1}^{D_1,s}_{D_2,\ldots,D_n}\ \&\ \cdots\ \&\ \overline{\pi_n.P_n}^{D_n,s}_{D_1,\ldots,D_{n-1}})\ ;\ \textsc{perform} - D_1\ \cdots\ - D_n))$, where D_1, ..., D_n are fresh predicates

  The use of & to simulate + is not obvious. We cannot use | to simulate + because a single-step communication in π-calculus has to be simulated using multiple reactor transition steps. We cannot use : to simulate + because : requires its two branches to terminate. We also cannot use | to simulate + because | causes one of the branches to commit too early.

  The locking trick is slightly modified below to quickly kill off dead branches in the above big interleaving.

- $\overline{x(\mathbf{y}).P}^{D_1,s}_{D_2,\ldots,D_n} = ((C(\mathbf{x}, \mathbf{y}, \mathbf{t}) \wedge t \neq s \wedge \neg D_1 \wedge \cdots \wedge \neg D_n \Rightarrow_1\ -C(x, y, t) + C'(x, y, t) + C''(x, y, t, s) + D_1);\ (C''(\mathbf{x}, \mathbf{y}, \mathbf{t}, s)\ \rightsquigarrow_1\ -C''(x, y, t, s)\ ;\ \overline{P})) \mid \neg C''(\mathbf{x}, \mathbf{y}, \mathbf{t}, s) \wedge (D_1\ \vee\ \cdots\ \vee\ D_n\ \Rightarrow_0\ !)$

- $\overline{\hat{x}\langle y\rangle.P}^{D_1,s}_{D_2,\ldots,D_n} = \text{PERFORM} \ + C(x,y,s)\ ;\ (((C'(x,y,s)\ \wedge\ \neg D_1\ \wedge\ \cdots\ \wedge\ \neg D_n \Rightarrow_1 +D_1 - C'(x,y,s))\ ;\ \overline{P})\ |\ (C(x,y,s) \wedge (D_1\ \vee\ \cdots\ \vee\ D_n)\ \Rightarrow_1\ -C(x,y,s)))$

Let us use the notation $r_1, \Delta_1 \longrightarrow^* \Delta_n, r_n$ to mean a sequence of reactor transitions $r_1, \Delta_1 \longrightarrow_{\delta_1} \Delta_2, r_2$, ..., $r_{n-1}, \Delta_{\delta_{n-1}} \longrightarrow_{\delta_{n-1}} \Delta_n, r_n$. We say that two sequences of such reactor transitions are equivalent if they start with the same reactor and state and end with the same reactor and state. Also, we say that a state Δ is proper if $\text{NOTED}(\Delta)$ and $\text{PENDING}(\Delta)$ are both empty and the only positive atomic fact entailed by $\text{FACTS}(\Delta)$ is $new(x)$ for some channel name x.

Lemma 1 *Let $P \leadsto Q$. Then there is Q' and proper states Δ and Δ' such that $Q \equiv Q'$ and $\overline{P}, \Delta \longrightarrow^* \Delta', \overline{Q'}$.*

Proof sketch. We proceed by induction on the derivation of $P \leadsto Q$. It is easy to see that each of the reaction rules, except the last one, has a canonical sequence of reactor transitions. Recall the last reactor rule that $P \leadsto Q$, if $P_0 \leadsto Q_0$ and $P_0 \equiv P$ and $Q_0 \equiv Q$. By induction hypothesis, we have $\overline{P_0}, \Delta \longrightarrow^* \Delta', \overline{Q_0'}$, $Q_0' \equiv Q_0 \equiv Q$. Since $P_0 \equiv P$, we can obtain the former from the latter by a series of structural congruences in our modified system, all of which have obvious counterparts in our reactor transitions. The theorem is proved as desired. $\Box$

Lemma 2 *Let $\overline{P}, \Delta \longrightarrow^* \Delta', \overline{Q}$ with Δ and Δ' both proper. Then $P \leadsto^* Q$.*

Proof. Without loss of generality, we assume that the sequence of reactor state transitions is as short as possible. The proof now proceeds by induction on the derivation of $\overline{P}, \Delta \longrightarrow^* \Delta', \overline{Q}$. There are two cases. The first case is when the sequence of state transitions is decomposable to $\overline{P}, \Delta \longrightarrow^* \Delta_0', \overline{P_0}$ followed by $\overline{P_0}, \Delta_0 \longrightarrow^* \Delta', \overline{Q}$. We conclude this case immediately by applying the induction hypothesis. The second case is when all intermediate reactors produced in the sequence of reactor state transitions are not derivable from any π-calculus expression. In this case, we use the assumption that the sequence of reactor state transitions is as short as possible to conclude that the sequence must correspond to a single application of a π-calculus reaction rule. $\Box$

Consequently, π-calculus can be embedded in our reactor language.