Concurrency and Communication in Transaction Logic

Anthony J. Bonner*
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4, Canada
bonner@db.toronto.edu

Michael Kifer†
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.
kifer@cs.sunysb.edu

Abstract

In previous work, we developed Transaction Logic (or $\mathcal{TR}$), which deals with state changes in deductive databases. $\mathcal{TR}$ provides a logical framework in which elementary database updates and queries can be combined into complex database transactions. $\mathcal{TR}$ accounts not only for the updates themselves, but also for important related problems, such as the order of update operations, non-determinism, and transaction failure and rollback. In the present paper, we propose *Concurrent Transaction Logic* (or $\mathcal{CTR}$), which extends Transaction Logic with connectives for modeling the concurrent execution of complex processes. Concurrent processes in $\mathcal{CTR}$ execute in an interleaved fashion and can communicate and synchronize themselves. Like classical logic, $\mathcal{CTR}$ has a "Horn" fragment that has both a procedural and a declarative semantics, in which users can program and execute database transactions. $\mathcal{CTR}$ is thus a deductive database language that integrates concurrency, communication, *and* updates. All this is accomplished in a completely logical framework, including a natural model theory and a proof theory. Moreover, this framework is flexible enough to accommodate many different semantics for updates and deductive databases. For instance, not only can updates insert and delete tuples, they can also insert and delete null values, rules, or arbitrary logical formulas. Likewise, not only can databases have a classical semantics, they can also have the well-founded semantics, the stable-model semantics, etc. Finally, the proof theory for $\mathcal{CTR}$ has an efficient SLD-style proof procedure. As in the sequential version of the logic, this proof procedure not only finds proofs, it also *executes* concurrent transactions, finds their execution schedules, and updates the database. A main result is that the proof theory is sound and complete for the model theory.

Appears in *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*, September 2–6 1996 Bonn, Germany, pages 142–156. MIT Press.

This and related papers and software are available at the following web page:
http://www.cs.toronto.edu/~bonner/transaction-logic.html

*Work supported in part by an Operating Grant from the Natural Sciences and Engineering Research Council of Canada.
†Supported in part by the NSF Grant IRI-9404629.

1 Introduction

Updates are a crucial component of any database programming language. Even the simplest database operations, such as withdrawal from a bank account, require updates. Unfortunately, updates are not accounted for by the classical Horn semantics of deductive databases. To fill this theoretical gap, we developed Transaction Logic (or $\mathcal{TR}$), an extension of first-order classical logic that provides a clean, logical account for state changes in databases and logic programs [8, 6, 9]. Like classical logic, $\mathcal{TR}$ has a "Horn" fragment that has both a procedural and a declarative semantics, in which users can program and execute database transactions. $\mathcal{TR}$ accounts not only for the updates themselves, but also for important related problems, such as the order of update operations, non-determinism, and transaction failure and rollback. All this is accomplished in a *completely logical framework*, including a natural model theory and a sound-and-complete proof theory.

One of the difficulties posed by updates in logic programs is that the *order* of update operations is often crucial. For example, erasing a database relation and inserting tuples into the relation are conflicting operations: the final outcome depends on which operation is executed first. The classical semantics of Horn logic does not address this issue. To deal with it, Transaction Logic extends classical logic with an operator—called *serial conjunction*—that allows a user to specify a linear order on a set of transactions. Using this operator in combination with logical rules, a user builds complex transactions out of simpler ones. This approach provides more flexibility and power than most other declarative database transaction languages. For instance, many such languages do not support subtransactions, transaction failure, or transaction rollback. Moreover, whereas most database transaction languages express all the database transactions computable in PSPACE, $\mathcal{TR}$ expresses all the transactions computable in EXPTIME (in the function-free case) [10].

In this paper, we extend Transaction Logic by allowing updates to be combined *concurrently*, as well as serially. We call the resulting logical system *Concurrent Transaction Logic* (or $\mathcal{CTR}$). Like Transaction Logic, $\mathcal{CTR}$ is a language for programming database transactions and applications.[1] Concurrency increases the flexibility, performance, and power of the language. Flexibility is increased since users are no longer forced to specify a total linear order on transactions. This opens the logic to a new range of advanced applications, including workflow management and multi-agent planning. Performance is increased since in a multi-user environment, a database management system (DBMS) can execute any concurrent process whose data items are available. It does not have to delay process execution because of an artificially-imposed linear order. Finally, power is increased since the logic can now express any computable database transaction [10], *i.e.*, concurrency increases expressibility from EXPTIME to RE (in the function-free case).

Although there has been considerable research on concurrency in databases, logic programming, and elsewhere, $\mathcal{CTR}$ is the only deductive database language that integrates concurrency, communication, *and* database updates in a completely logical framework. Such integration presents interesting new possibilities for the programmer. For instance, concurrent processes can now communicate via the database; that is, one process can read what another process has written. This form of communication leads to a programming style that is very different from that of existing concurrent logic programming (CLP) languages, as illustrated in this paper.

In CLP languages [32], concurrent processes communicate via shared variables and unification. This kind of communication is orthogonal to communication via the database. Both are possible in $\mathcal{CTR}$. Implementations of $\mathcal{CTR}$ may therefore adopt many of the techniques of shared-variable communication developed for CLP. However, this possibility is *not* the focus of this paper. Instead, we focus on concurrent processes that interact and communicate via the database. Indeed, one of the novelties of $\mathcal{CTR}$ is that it provides a logical foundation for exactly this kind of interaction.

Programming with states is very common, and the practice of logic programming shows that it is also unavoidable. Unfortunately, classical logic is stateless. One way to get around this problem is to represent states as terms, as in the situation calculus [21, 31]. However, this approach has not taken roots in logic programming for many reasons, including the complexity of dealing with the frame problem [9]. In Prolog, efficient techniques have been developed where states are represented as lists or terms, which do not require frame axioms. However, these techniques rely heavily on Prolog's control strategy, cuts, or on other non-logical features. Moreover, it is not clear that this approach is practical for very large states,

[1] Consequently, this paper is about database application programming, *not* concurrency control and recovery.

such as databases. Only "tiny" databases fit in main memory, and databases several gigabytes in size are commonplace. Can Prolog efficiently manage a 10 GByte list? What if several processes need to modify the list concurrently?

Another possibility is to do updates via the database. Prolog even supplies primitive operators for doing this, such as *assert* and *retract*. Unfortunately, these update operators have no logical semantics, and each time a programmer uses them, he moves further away from declarative programming. Moreover, Prolog programs using these operators are often awkward and the most difficult to understand, debug and maintain. These problems are all exacerbated by concurrency.

Another problem is that updates in Prolog are not integrated into a complete logical system. Thus, it is not clear how *assert* and *retract* should interact with other logical operators. For instance, what do the following formulas mean: $assert(X) \vee assert(Y)$, $\forall X \exists Y\, assert(p(X,Y))$, $assert(X) \leftarrow retract(Y)$, or $\neg assert(X)$. Also, how does one logically account for the fact that the order of updates is important? Finally, what does it mean to update a database that contains arbitrary logical formulas? None of these questions is addressed by Prolog's classical semantics or its update operators. Likewise, these problems are not addressed by existing concurrent logic programming languages [32]. In fact, the problems of updates are even more complex in a concurrent environment.

Concurrent Transaction Logic provides a general solution to the aforementioned limitations, both of Prolog and of concurrent logic programming languages. Intuitively, a $\mathcal{CTR}$ program consists of a number of concurrent processes, where each process produces a sequence of elementary database operations. By interleaving these sequences, we obtain a new sequence of operations, which can then be executed. The set of possible interleavings is determined by interactions between the processes. For instance, if one process writes data that another process must read, then the write operation must come *before* the read operation in the interleaved sequence. These interactions are specified by $\mathcal{CTR}$ programs. Given such a program, the model theory describes the set of all possible interleavings, and the proof theory generates them.

From the programmer's point of view, $\mathcal{CTR}$ supplies a syntax and a semantics by which elementary database operations can be combined serially and concurrently into complex programs. Like sequential Transaction Logic, the emphasis in $\mathcal{CTR}$ is not on specifying elementary operations, but on combining them logically into programs. This emphasis reflects programming practice. In databases and logic programming (as in C and Pascal), application programmers spend little if any time specifying elementary operations, but a lot of time combining them into complex transactions and programs.

Nevertheless, the set of elementary operations is an important feature of a language, and can determine its domain of application. Moreover, practice shows that elementary operations can vary widely. For example, in C and Pascal, changing the value of a variable is an elementary operation. In Prolog, asserting or retracting a clause is elementary. In database applications, SQL statements are the basic building blocks. In scientific and engineering programs, basic operations include Fourier transforms, matrix inversion, least-squares fitting, and operations on DNA sequences [14, 20]. In workflow management systems, elementary operations can include any number of application programs and legacy systems [4]. In all cases, the elementary operations are building blocks from which larger programs and software systems are built.

Although elementary operations vary dramatically, the logic for combining them does not. In fact, the same control features arise over-and-over again. These features include sequential and concurrent composition, iterative loops, conditionals, subroutines and recursion. $\mathcal{CTR}$ provides a logical framework in which these control features can be expressed. This framework is *orthogonal* to the elementary operations. $\mathcal{CTR}$ can therefore be used with *any* set of elementary database operations, including destructive updates. To achieve this flexibility, $\mathcal{CTR}$ treats a database as a collection of abstract datatypes, each with its own special-purpose access methods. These methods are provided to $\mathcal{CTR}$ as elementary operations, and they are combined by $\mathcal{CTR}$ programs into complex transactions. This approach separates the specification of elementary operations from the logic of combining them. As we shall see, this separation has two main benefits: (i) it allows us to develop a logic language for programming state-changing procedures without committing to a particular theory of elementary updates; and (ii) it allows $\mathcal{CTR}$ to accommodate a wide variety of database semantics, from classical to non-monotonic to various other non-standard logics. In this way, $\mathcal{CTR}$ provides the logical foundations for extending the logic-programming paradigm to a host of new applications in which a given set of operations must be combined into larger programs or software systems.

2 Syntax

2.1 Transaction Formulas

The alphabet of a $\mathcal{CTR}$ language $\mathcal{L}$ includes three countable sets of symbols: a set $\mathcal{F}$ of function symbols, a set $\mathcal{V}$ of variables, and a set $\mathcal{P}$ of predicate symbols. Each function and predicate symbol has an associated *arity*, indicating how many arguments the symbol takes. Constants are viewed as 0-ary function symbols, and propositions are viewed as 0-ary predicate symbols. The language $\mathcal{L}$ also includes the logical connectives $\wedge$, $\otimes$, $\mid$, $\neg$, the modal operator $\odot$, and the quantifier $\forall$. Function terms are defined as usual in first-order logic.

As the alphabet suggests, transaction formulas extend first-order formulas with two new connectives, $\otimes$ and $\mid$, which we call *serial conjunction* and *concurrent conjunction*, respectively, and one modal operator $\odot$ for specifying atomic actions. The simplest transaction formulas are *atomic formulas*, which are defined as in classical logic via predicates and terms. In addition, if ϕ and ψ are transaction formulas, then so are $\phi \wedge \psi$, $\phi \otimes \psi$, $\phi \mid \psi$, $\neg\phi$, $\odot\,\phi$, and $(\forall X)\phi$, where X is a variable. Intuitively, the formula $\phi \otimes \psi$ means that the subtransactions ϕ and ψ execute serially, *i.e.*, first ϕ executes to completion, and then ψ executes. The formula $\phi \mid \psi$ means that the subtransactions ϕ and ψ execute concurrently. The formula $\odot\,\phi$ means that ϕ must execute "atomically", meaning that its execution *should not* be interleaved with other transactions. Note that classical first-order formulas are transaction formulas that do not use the connectives $\otimes$, $\mid$, or the modal operator $\odot$. As in classical logic, we introduce convenient abbreviations for complex formulas. For instance, $\phi \leftarrow \psi$ is an abbreviation for $\phi \vee \neg\psi$. Likewise, $\phi \vee \psi$ is an abbreviation for $\neg(\neg\phi \wedge \neg\psi)$, and $\exists\,\phi$ is an abbreviation for $\neg\forall\neg\phi$. Other useful abbreviations are developed in [9]. Here are three examples of transaction formulas: $(a \otimes b) \mid \odot\,(c \otimes d)$, $a(X) \leftarrow [b(X) \mid c(X,Y)]$, $\forall X[a(X) \vee \neg b(X) \otimes \neg c(X,Y)]$

2.2 Database States, Elementary Operations, and Oracles

Logical theories always come with a parameter: a language for constructing well-formed formulas. This language is not fixed, since almost any set of constants, variables and predicate symbols can be "plugged into" it. In addition, $\mathcal{CTR}$ theories have another parameter: a pair of oracles that encapsulate elementary database operations. Like the language of the logic, the oracles are not fixed, since almost any pair of oracles can be "plugged into" a $\mathcal{CTR}$ theory. The oracles come with a set of *database states*, upon which they operate. In practice, we expect that each database state will be a set of data items, as in the theory of transaction management [2]. Intuitively, a data item can be any kind of persistent object, such as a tuple, a disk page, an email queue, or a logical formula. Formally, however, a database state has no structure, and our only access to it is through the two oracles.

Both oracles are mappings. The *data oracle*, $\mathcal{O}^d$, is a mapping from states to sets of first-order formulas. Intuitively, if $\mathbf{D}$ is a state, then $\mathcal{O}^d(\mathbf{D})$ is the set of formulas that are true of the state. Likewise, the *transition oracle*, $\mathcal{O}^t$, is a mapping from pairs of states to sets of ground atomic formulas.[2] Intuitively, if $b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ then b is an elementary update that changes state $\mathbf{D}_1$ into state $\mathbf{D}_2$. Primitive data access is thus specified outside $\mathcal{CTR}$, by using the oracles. The oracles also provide a semantics for the data items: the data oracle provides a *static* semantics; and the transition oracle provides a *dynamic* semantics, that is, a semantics of change. By using oracles in this way, $\mathcal{CTR}$ can accommodate different data access primitives and different database semantics.

Here are some examples of data and transition oracles. Some of these can be combined to yield more powerful oracles. Typically, such combinations are possible when oracles operate on disjoint domains of data items.

- *Relational Oracles:* A state is a set of ground atomic formulas, $\mathbf{D}$. The data oracle simply returns all these formulas. Thus, $\mathcal{O}^d(\mathbf{D}) = \mathbf{D}$. Moreover, for each predicate symbol p in $\mathbf{D}$, the transition oracle defines two new predicates, $p.ins$ and $p.del$, representing the insertion and deletion of single atoms, respectively. Formally, $p.ins(\overline{x}) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_2 = \mathbf{D}_1 \cup \{p(\overline{x})\}$. Likewise, $p.del(\overline{x}) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_2 = \mathbf{D}_1 - \{p(\overline{x})\}$. SQL-style bulk updates can also be defined by the transition oracle [9], as can primitives for creating new constant symbols.

[2] Ground atomic formulas are required for simplicity.

- *Horn Oracles:* A state $\mathbf{D}$ is a set of Horn rules and $\mathcal{O}^d(\mathbf{D})$ is the minimal Herbrand model of $\mathbf{D}$. One may also want to augment $\mathcal{O}^d(\mathbf{D})$ with the rules from $\mathbf{D}$. The transition oracle can be defined to insert and delete Horn rules from $\mathbf{D}$.

- *Communication Oracles:* A state is a set of message queues, each representing a communication channel. Each queue has a name, q, and a list of messages. The data oracle defines a binary predicate, *peek*, which corresponds to reading a message without consuming it. Formally, $peek(q, msg) \in \mathcal{O}^d(\mathbf{D})$ iff the front of queue q in $\mathbf{D}$ has the message *msg*. The transition oracle defines two binary predicates, *send* and *receive*, which correspond to sending and receiving messages along a communication channel. Formally, $send(q, msg) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_2$ is obtained from $\mathbf{D}_1$ by adding *msg* to the end of queue q. Likewise, $receive(q, msg) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff the front of queue q in $\mathbf{D}_1$ has the message *msg*, and $\mathbf{D}_2$ is obtained from $\mathbf{D}_1$ by removing *msg* from the front of q. Intuitively, *receive* reads a message on channel q and consumes it. Other communication primitives, such as creating and destroying channels, can also be defined by the oracles.[3]

Unlike transaction formulas, we do not expect the oracles to be coded by casual users. Although the oracles allow for many different semantics, we envision that any logic programming system based on $\mathcal{CTR}$ will provide a carefully selected repertoire of built-in database semantics and a tightly controlled mechanism for adding new ones. This latter mechanism would not be available to ordinary programmers. For this reason, we assume in this paper that the data and transition oracles are fixed. A more complete discussion of data and transition oracles can be found in [9].

2.3 Examples

This section gives some simple examples of Concurrent Transaction Logic programs, or *transaction bases*. A transaction base is a finite set of transaction formulas. In this paper, all examples of transaction bases use so-called *concurrent Horn rules*, which are defined formally in Section 4. The body of these rules consists of atomic formulas connected by serial or concurrent conjunction. Atomic modalities are also allowed. Each atomic formula stands for a query, an update or, more generally, a transaction. For example, the formula $a \leftarrow (b_1 \otimes b_2) \mid (c_1 \otimes c_2)$ is a concurrent Horn rule. Intuitively, it says, "To execute transaction a, execute concurrently the two transactions $b_1 \otimes b_2$ and $c_1 \otimes c_2$. To execute $b_1 \otimes b_2$, first do b_1 then b_2, and similarly for $c_1 \otimes c_2$." The rule-head, therefore, acts as the name of a transaction, while the rule-body acts as the transaction definition.

All examples in this paper are based on a combination of the relational and communication oracles defined in Section 2.2. A state is a collection of ground atomic formulas and communication channels (queues). A relation p is accessed by using the predicates p, $p.ins$, and $p.del$. Communication takes place using the predicates *send*, *receive*, and *peek*. Formally, these predicates are no different from other predicates, and their fancy names is a mere convention. Their special status, however, comes from the choice of the data and transition oracles.

The first example, below, focuses on serial conjunction and shows how updates can be combined with queries to define complex transactions. It also illustrates the use of transaction subroutines (or nested transactions), and shows how *sequential* Transaction Logic improves upon Prolog's update operators.

Example 2.1 (Financial Transactions) Suppose the balance of a bank account is given by the relation $balance(Acct, Amt)$. We define four transactions: $change\_balance(Acct, Bal1, Bal2)$, to change the balance of an account from one amount to another; $withdraw(Amt, Acct)$, to withdraw an amount from an account; $deposit(Amt, Acct)$, to deposit an amount into an account; and $transfer(Amt, Acct1, Acct2)$, to transfer an amount from one account to another. These transactions are defined by the following four rules, which form a transaction base:

[3] It is important to keep in mind here that concurrency and communication are *not* built into the oracles in $\mathcal{CTR}$. Communication oracles are just a matter of convenience. We shall see that all communication primitives can be expressed using $\mathcal{CTR}$'s logical connectives.

$$
\begin{aligned}
&transfer(Amt, Acct1, Acct2) \leftarrow \odot\,[withdraw(Amt, Acct1) \otimes deposit(Amt, Acct2)]\\
&withdraw(Amt, Acct) \leftarrow balance(Acct, Bal) \otimes Bal \geq Amt \otimes change\_balance(Acct, Bal, Bal - Amt)\\
&deposit(Amt, Acct) \leftarrow balance(Acct, Bal) \otimes change\_balance(Acct, Bal, Bal + Amt)\\
&change\_balance(Acct, Bal1, Bal2) \leftarrow balance.del(Acct, Bal1) \otimes balance.ins(Acct, Bal2)
\end{aligned}
$$

In each rule, the premises are evaluated from left to right—an evaluation order imposed by the serial conjunction. For instance, the first rule says: to transfer an amount, Amt, from $Acct1$ to $Acct2$, first withdraw Amt from $Acct1$ and, if the withdrawal succeeds, deposit Amt in $Acct2$. Likewise, the second rule says, to withdraw Amt from an account $Acct$, first retrieve the balance of the account; then check that the account will not be overdrawn by the transaction; if all is well, change the balance from Bal to $Bal - Amt$. Notice that the atom $balance(Acct, Bal)$ is a query that retrieves the balance of the specified account, and $Bal \geq Amt$ is a test. All other atoms in this example are updates. The last rule changes the balance of an account by deleting the old balance and then inserting the new one. Unlike the other rules, the last rule is defined via built-in, elementary updates, $balance.del$ and $balance.ins$. Finally, observe that the entire transfer transaction is atomic, as specified in the body of the first rule. Thus, if two transfers are carried out concurrently, then they will not interfere with each other. □

Observe that the rules in Example 2.1 can be cast into Prolog syntax by replacing "$\otimes$" with "," and replacing the elementary transitions, $balance.ins$ and $balance.del$, with *assert* and *retract*, respectively. However, the resulting, apparently innocuous, Prolog program does not behave correctly! The problem is that Prolog does not undo updates during backtracking. As an example, consider a transaction involving two transfers, defined as follows:

$$?-\ transfer(Fee, Client, Broker) \otimes transfer(Cost, Client, Seller)$$

That is, a fee is transferred from a client to a broker, and then a cost is transferred from the client to a seller. Because this is intended to be a transaction, it must behave *atomically*; that is, it must execute entirely or not at all. Thus, if the second transfer fails, then the first one must be rolled back. In this respect, $\mathcal{CTR}$ behaves correctly. Prolog, however, does not, since it commits updates immediately and does not undo partially executed transactions. Thus, if the second transfer above were to fail (say, because the client's account would be overdrawn by the transaction), then Prolog would *not* undo the first one, thus leaving the database in an inconsistent state.

The non-logical behavior of Prolog updates is notorious for making Prolog programs cumbersome and heavily dependent on Prolog's backtracking strategy. Some Prologs do have the means to implement backtrackable updates through the use of non-logical operations, such as *assert*, *retract*, and cut (!). However, the logic behind the resulting update operations has not been developed. Transaction Logic can be considered to provide the missing semantics and proof theory, although there are many fine differences between simplistic implementations of backtrackable updates and our logic.

The above example illustrates some salient features of sequential Transaction Logic, but it consists of a single process that executes alone. Here is a simple example of two processes executing concurrently:

$$?-\ transfer(Fee, Client, Broker) \mid transfer(Cost, Client, Seller)$$

Once again, because this is a transaction, it must behave *atomically*. Thus, if one transfer process fails, then the other one must be rolled back. The semantics of Concurrent Transaction Logic specifies exactly this. This illustrates in the simplest way the combination of concurrency and updates that $\mathcal{CTR}$ supports. The next examples illustrate how $\mathcal{CTR}$ also supports communication. The first example shows how processes in Concurrent Transaction Logic can synchronize themselves by exchanging messages along communication channels.

Example 2.2 (Synchronization) Consider the following transaction base, which defines two processes, $processA$ and $processB$:

$$
\begin{aligned}
processA &\longleftarrow taskA_1 \otimes send(ch_1, startB_2) \otimes taskA_2 \otimes receive(ch_2, startA_3) \otimes taskA_3\\
processB &\longleftarrow taskB_1 \otimes receive(ch_1, startB_2) \otimes taskB_2 \otimes send(ch_2, startA_3) \otimes taskB_3
\end{aligned}
$$

Processes *processA* and *processB* invoke three tasks each. The concurrent transaction $?-\ processA \mid processB$ causes *processA* and *processB* to execute concurrently while synchronizing the execution of their tasks by passing messages along channels ch_1 and ch_2. In particular, $taskB_2$ cannot start executing until $taskA_1$ is finished, and $taskA_3$ cannot start executing until $taskB_2$ is finished.

While executing $taskA_1$ and $taskB_1$, the two processes run concurrently, without interacting with each other. However, when *processB* completes $taskB_1$, it cannot start $taskB_2$ until it receives a message from *processA*. To understand why, recall the definition of the communication oracle. The transition $receive(ch_1, startB_2)$ cannot execute (which means it cannot be true on any execution path emanating from the current state—see Section 3) unless the message $startB_2$ is sitting at the front of the queue ch_1. However, this message will not be there unless *processA* makes the transition $send(ch_1, startB_2)$, which will only happen after it executes $taskA_1$. In this way, *processB* is synchronized with *processA*.

In a similar fashion, on completing $taskA_2$, *processA* cannot start $taskA_3$ until it receives a message from *processB*, which will only happen after *processB* executes $taskB_2$. Note that $\mathcal{CTR}$ processes do not need a hand-shake in order to communicate (but hand-shake can be always achieved by sending more messages). $\Box$

Example 2.3 (Producer/Consumer) Consider the following transaction base that defines two processes:

$$
\begin{array}{l}
produce([N|List]) \longleftarrow send(ch, N) \otimes produce(List) \\
produce([\,]) \longleftarrow send(ch, done) \\[4pt]
consume(Sum, Total) \longleftarrow receive(ch, N) \otimes consume(Sum + N, Total) \\
consume(Total, Total) \longleftarrow receive(ch, done)
\end{array}
$$

When executed concurrently, the producer sends a list of numbers to the consumer, one number at a time. As the consumer receives the numbers, it adds them together in a running sum (the first argument of *consume*). After the producer has sent all the numbers in the list, it sends the message *done* to the consumer. When the consumer receives the *done* message, it terminates its recursion and returns the total sum (the second argument of *consume*). For instance, the transaction

$$?-\ \ produce([1, 2, 3, 4, 5]) \ \mid consume(0, Total)$$

causes the producer to send the numbers 1 through 5 to the consumer, while the consumer concurrently receives them, sums them up, and returns $Total = 15$. $\Box$

Section 2.2 described the predicates for sending and receiving messages via oracles. However, it is worth mentioning that in $\mathcal{CTR}$ they can also be defined via run-of-the-mill insert and delete primitives for ground atoms. The three rules below implement a simplified version of these predicates, in which a communication channel is a *set* of messages, rather than a *queue*. Intuitively, these rules treat a channel as a "pool" of messages, rather than a "stream." Queues and queue operations are also easily implemented in terms of insert and delete primitives.

$$
\begin{array}{ll}
send(Ch, Msg) \longleftarrow channel.ins(Ch, Msg) & \%\ Transmit. \\
peek(Ch, Msg) \longleftarrow channel(Ch, Msg) & \%\ Listen,\ do\ not\ consume. \\
receive(Ch, Msg) \longleftarrow \odot\ (channel(Ch, Msg) \otimes channel.del(Ch, Msg)) & \%\ Listen\ and\ consume.
\end{array}
$$

In these rules, communication channels are implemented as a binary database relation, *channel*. Intuitively, $channel(Ch, Msg)$ means that channel Ch contains message Msg. Sending a message along a channel amounts to inserting a tuple in the *channel* relation, peeking at a message amounts to retrieving a tuple, and receiving a message amounts to deleting a tuple. Receiving a message requires that the message be in the channel, so tuple deletion is preceded by a query, to make sure that the tuple is there. Note the use of the modal operator $\odot$, which ensures that checking for a message and consuming it is done atomically, preventing other processes from interposing their actions between the test $channel(Ch, Msg)$ and the update $channel.del(Ch, Msg)$.

3 Model Theory

The semantics of the sequential Transaction Logic is based on sequences of database states, called *paths* [9, 8, 6]. Concurrent Transaction Logic extends this idea to sequences of paths, called *multi-paths*, or *m-paths*. Multi-paths provide the basis for a logical semantics of concurrent conjunction. This section describes the intuitive motivation for multi-paths, and develops them into a formal model theory of $\mathcal{CTR}$.

3.1 Multi-Paths

When a user executes a transaction, the database may change, going from its initial state to some final state. In doing so, the database may pass through any number of intermediate states. For example, the transaction $\phi = a.ins \otimes b.ins \otimes c.ins$ takes the database from an initial state, $\mathbf{D}$, through the intermediate states $\mathbf{D} + \{a\}$ and $\mathbf{D} + \{a, b\}$, to the final state $\mathbf{D} + \{a, b, c\}$. In sequential Transaction Logic [9, 6], a finite sequence of states is called a *path*, and transactions are true on paths. For example, transaction ϕ is true on the path $\langle\{\}\ \{a\}\ \{a,b\}\ \{a,b,c\}\rangle$, in which the initial state is the empty database. We also say that $\langle\{\}\ \{a\}\ \{a,b\}\ \{a,b,c\}\rangle$ is an *execution path* of ϕ. Execution paths allow us to model a wide range of dynamic constraints [9, 7, 11].

To model concurrent processes, we generalize the notion of path to *multi-path*, or *m-path*. An m-path records the execution history of a process. Intuitively, it represents periods of continuous execution, separated by periods of suspended execution (during which other processes may execute). Formally, an *m-path* is a finite sequence of paths, where each constituent path represents a period of continuous execution. For example, if $\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_8$ are database states, then $\langle \mathbf{D}_1\mathbf{D}_2\mathbf{D}_3,\ \mathbf{D}_4\mathbf{D}_5,\ \mathbf{D}_6\mathbf{D}_7\mathbf{D}_8\rangle$ is an m-path. If this m-path represents the execution history of process ϕ, then the process has three periods of continuous execution. In the first period, ϕ changes the database from $\mathbf{D}_1$ to $\mathbf{D}_2$ to $\mathbf{D}_3$, after which ϕ is suspended and re-awakened when the database is at state $\mathbf{D}_4$. At this point, ϕ starts its the second period of continuous execution, and changes the database from $\mathbf{D}_4$ to $\mathbf{D}_5$. At state $\mathbf{D}_5$, ϕ is suspended once again and re-awakened when the database reaches state $\mathbf{D}_6$ (due to other processes). This begins the third period of ϕ's execution, which changes the database from $\mathbf{D}_6$ to $\mathbf{D}_7$ to $\mathbf{D}_8$.

Note that paths are a special case of m-paths, as can be seen from our notation.

Operations on Multi-Paths

Concurrent Transaction Logic extends first-order logic with three logical operators: serial conjunction, $\otimes$, concurrent conjunction, $\mid$, and the modality of atomicity, $\odot$. Semantically, these three operators are related to three kinds of operations on m-paths: *concatenation*, *interleaving*, and *reduction*. The rest of this subsection defines these three operations.

Definition 3.1 (Concatenation) Suppose that $\kappa = \langle \mathbf{D}_1\ \ldots\ \mathbf{D}_k\rangle$ and $\kappa' = \langle \mathbf{D}_k\ \ldots\ \mathbf{D}_{k+l}\rangle$ are two *paths*, where $\mathbf{D}_k$ is the last state in κ and also the first state in κ'. Then, their *concatenation* is the path $\kappa \circ \kappa' = \langle \mathbf{D}_1\ \ldots\ \mathbf{D}_k\ \ldots\ \mathbf{D}_{k+l}\rangle$. If $\tau = \langle \kappa_1, \ldots, \kappa_n\rangle$ and $\tau' = \langle \kappa'_1, \ldots, \kappa'_m\rangle$ are two *m-paths*, then *their* concatenation is the m-path $\tau \bullet \tau' = \langle \kappa_1, \ldots, \kappa_n, \kappa'_1, \ldots, \kappa'_m\rangle$. □

The second operation on m-paths is interleaving. To illustrate, suppose that m-paths $\langle\kappa_1, \kappa_2\rangle$ and $\langle\kappa'_1, \kappa'_2\rangle$ represent executions of processes ϕ and ϕ', respectively. If we interleave these two m-paths, then we get new m-paths representing interleaved executions of ϕ and ϕ', that is, executions of the composite process $\phi \mid \phi'$. For example, the following four m-paths represent executions of $\phi \mid \phi'$: $\langle\kappa_1, \kappa'_1, \kappa_2, \kappa'_2\rangle$, $\langle\kappa'_1, \kappa_1, \kappa'_2, \kappa_2\rangle$, $\langle\kappa_1, \kappa'_1, \kappa'_2, \kappa_2\rangle$, $\langle\kappa'_1, \kappa_1, \kappa_2, \kappa'_2\rangle$.

Definition 3.2 (Interleaving) If π and $\pi_1, ..., \pi_n$ are m-paths (*i.e.*, sequences of paths), then π is an *interleaving* of $\pi_1, ..., \pi_n$ if π can be partitioned into order-preserving subsequences $\mathcal{C}_1, ..., \mathcal{C}_n$, such that each $\mathcal{C}_i$ is π_i. The set of all interleavings of π_1 and π_2 is denoted $\pi_1 \| \pi_2$. □

The final operation on m-paths is reduction. The idea is that if the database state does not change while a process is suspended, then the process can also execute continuously, without any suspension. For instance, if a process can execute along the m-path $\langle \mathbf{D}_1\mathbf{D}_2,\ \mathbf{D}_2\mathbf{D}_3\rangle$, then the process is suspended in state $\mathbf{D}_2$ and re-awakened in state $\mathbf{D}_2$. It should therefore be able to execute continuously along the path $\langle \mathbf{D}_1\mathbf{D}_2\mathbf{D}_3\rangle$. We say that the former m-path reduces to the latter.

Definition 3.3 (Reduction) Let $\tau = \langle \kappa_1, \ldots, \kappa_n \rangle$ be an m-path, where each κ_i is a path. If the paths κ_i and κ_{i+1} can be concatenated, for some i, then we say that τ *reduces* to $\tau' = \langle \kappa_1, \ldots, \kappa_{i-1}, \kappa_i \circ \kappa_{i+1}, \kappa_{i+2}, \ldots, \kappa_n \rangle$. We also require reduction to be closed under transitivity and reflection. □

Observe that reduction is not reversible, since some processes may be specified as atomic. An atomic process must execute continuously, without being suspended and interleaved with other processes. For instance, an atomic process that can execute along $\langle \mathbf{D}_1\mathbf{D}_2\mathbf{D}_3\rangle$ *cannot* execute along $\langle \mathbf{D}_1\mathbf{D}_2,\ \mathbf{D}_2\mathbf{D}_3\rangle$. In general, atomic processes can only execute along paths (*i.e.*, without suspension).

3.2 Semantics

$\mathcal{CTR}$ formulas are interpreted by multi-path structures. A multi-path structure assigns a classical first-order structure to each m-path, specifying which atoms are true on what m-paths. In turn, these atoms determine which formulas are true on what m-paths. Intuitively, a formula that is true on an m-path represents an action that takes place along the m-path. Multi-path structures generalize the path structures of sequential Transaction Logic. Many subtle points about path structures apply equally well to multi-path structures, and the reader is referred to [9, 8, 6] for a thorough discussion. The main novelties of m-path structures are the m-paths themselves, the interleaving operation, and reduction, which account for concurrent and atomic processes.

Recall that $\mathcal{CTR}$ comes with a language, $\mathcal{L}$, and a pair of oracles, $\mathcal{O}^d$ and $\mathcal{O}^t$, which determine the syntax of formulas and the semantics of databases, as described in Section 2. In the rest of this paper, the language $\mathcal{L}$ and the oracles are implicit. Also, $\models^c$ denotes satisfaction in classical first-order models.

Definition 3.4 (Multi-Path Structures) Let $\mathcal{L}$ be a language of $\mathcal{CTR}$ with the set of function symbols $\mathcal{F}$. A *multi-path structure* (abbr. *m-path structure*) $\mathbf{M}$ over $\mathcal{L}$ is a triple $\langle U, I_{\mathcal{F}}, I_{path}\rangle$, where

- U is a set, called the *domain* of $\mathbf{M}$.
- $I_{\mathcal{F}}$ is an interpretation of function symbols in $\mathcal{L}$. It assigns a function $U^n \longmapsto U$ to every n-ary function symbol in $\mathcal{F}$.

  Given U and $I_{\mathcal{F}}$, let $Struct(U, I_{\mathcal{F}})$ denote the set of all classical first-order semantic structures over $\mathcal{L}$ of the form $\langle U, I_{\mathcal{F}}, I_{\mathcal{P}}\rangle$, where U is the domain of the structure, $I_{\mathcal{P}}$ is a mapping that interprets predicate symbols in $\mathcal{P}$ by relations on U, and U and $I_{\mathcal{F}}$ are the same as in $\mathbf{M}$.
- I_{path} is a total mapping from the m-paths in $\mathcal{L}$ to the semantic structures in $Struct(U, I_{\mathcal{F}})$. I_{path} is subject to the following restrictions:
  - *Conformance with m-path reduction*: If an m-path π_1 reduces to π_2, then $I_{path}(\pi_1) \models^c a$ implies $I_{path}(\pi_2) \models^c a$, for every atom a (*i.e.*, if a can execute along π_1, then it can execute along π_2).
  - *Compliance with the data oracle*: $I_{path}(\langle \mathbf{D}\rangle) \models^c \phi$ for every formula $\phi \in \mathcal{O}^d(\mathbf{D})$.
  - *Compliance with the transition oracle*: $I_{path}(\langle \mathbf{D}_1\ \mathbf{D}_2\rangle) \models^c b$ for every atom $b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$. □

As in classical logic, a *variable assignment*, ν, is a mapping $\mathcal{V} \longmapsto U$ that takes variables as input and returns domain elements as output. We extend the mapping from variables to terms in the usual way.

Definition 3.5 (Satisfaction) Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{path}\rangle$ be an m-path structure, let π be an arbitrary m-path, and let ν be a variable assignment. Then,

1. **Base Case:** $\mathbf{M}, \pi \models_\nu p(t_1, \ldots, t_n)$ if and only if $I_{path}(\pi) \models^c_\nu p(t_1, \ldots, t_n)$, for any atomic formula $p(t_1, \ldots, t_n)$.

2. **Negation:** $\mathbf{M}, \pi \models_\nu \neg\phi$ if and only if it is not the case that $\mathbf{M}, \pi \models_\nu \phi$.
3. **"Classical" Conjunction:** $\mathbf{M}, \pi \models_\nu \phi \wedge \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ and $\mathbf{M}, \pi \models_\nu \psi$.
4. **Serial Conjunction:** $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ if and only if $\mathbf{M}, \pi_1 \models_\nu \phi$ and $\mathbf{M}, \pi_2 \models_\nu \psi$ for *some* m-paths π_1, π_2 whose concatenation $\pi_1 \bullet \pi_2$ reduces to π.
5. **Concurrent Conjunction:** $\mathbf{M}, \pi \models_\nu \phi \mid \psi$ if and only if $\mathbf{M}, \pi_1 \models_\nu \phi$ and $\mathbf{M}, \pi_2 \models_\nu \psi$ for *some* m-paths π_1, π_2 with an interleaving in $\pi_1 \| \pi_2$ that reduces to π.
6. **Universal Quantification:** $\mathbf{M}, \pi \models_\nu \forall X.\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *every* variable assignment μ that agrees with ν everywhere except on X.
7. **Atomic Processes:** $\mathbf{M}, \pi \models_\nu \odot\phi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ and π is a path (not a general m-path).

As in classical logic, the mention of variable assignment can be omitted for *sentences*, *i.e.*, for formulas with no free variables. From now on, we will deal only with sentences, unless explicitly stated otherwise. If $\mathbf{M}, \pi \models \phi$, then we say that ϕ is *satisfied* (or is *true*) on m-path π in structure $\mathbf{M}$. □

It can be seen from the definitions that on paths of length 1, the connectives $\otimes$, $|$, and $\wedge$ all reduce to the usual conjunction of classical predicate logic. Thus, the "classical," serial, and concurrent conjunctions in $\mathcal{CTR}$ extend the usual conjunction in classical logic from states to execution paths, albeit in three different ways. The following lemma states a basic property of satisfaction on m-paths. It is proved by a straightforward induction on the structure of ψ.

Lemma 3.6 *If m-path π_1 reduces to π_2, then for any m-path structure M and transaction formula ψ,*

$$\textit{if} \quad M, \pi_1 \models \psi \quad \textit{then} \quad M, \pi_2 \models \psi$$

Definition 3.7 (Models) An m-path structure $\mathbf{M}$ is a *path-model* (or simply a *model*) of a transaction formula ϕ, written $\mathbf{M} \models \phi$, if and only if $\mathbf{M}, \pi \models \phi$ for every m-path π. An m-path structure is a model of a set of formulas if and only if it is a model of every formula in the set. □

3.3 Execution as Entailment

A $\mathcal{CTR}$ program has two distinct parts: a transaction base $\mathbf{P}$ and an initial database state $\mathbf{D}$. Recall that the database is a set of data items, and the transaction base is a finite set of transaction formulas. Of these two parts, only the database is updatable. The transaction base is immutable and specifies procedures for updating the database and answering queries. The transaction base will normally be composed of formulas containing the new connectives, $\otimes$, $|$, and $\odot$, although classical first-order formulas are also allowed. With this in mind, we are ready to define *executional entailment*, a concept that provides a logical account of transaction execution.

Definition 3.8 (Executional Entailment) Let $\mathbf{P}$ be a transaction base. If ϕ is a transaction formula and $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ is a sequence of database states, then the statement

$$\mathbf{P}, \mathbf{D}_0\ \mathbf{D}_1\ \ldots\ \mathbf{D}_n \models \phi \tag{1}$$

is true if and only if $\mathbf{M}, \langle \mathbf{D}_0\ \mathbf{D}_1\ \ldots\ \mathbf{D}_n \rangle \models \phi$, for every model $\mathbf{M}$ of $\mathbf{P}$.[4] Related to this is the statement

$$\mathbf{P}, \mathbf{D}_0 \text{---} \models \phi \tag{2}$$

which are true iff Statement (1) is true for some sequence of database states $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$. □

[4]Being parameters to $\mathcal{CTR}$, the oracles and hence the set of all states are fixed.

Intuitively, Statement (1) says that a successful execution of transaction ϕ can change the database from state $\mathbf{D}_0$ to $\mathbf{D}_1 \ldots$ to $\mathbf{D}_n$. Formally, it means that in every model of $\mathbf{P}$, the m-path $\langle \mathbf{D}_0\ \mathbf{D}_1\ \ldots\ \mathbf{D}_n \rangle$ satisfies formula ϕ. The statement is read as follows: "Under the transaction base $\mathbf{P}$, transaction ϕ may transform database $\mathbf{D}_0$ into database $\mathbf{D}_n$ by passing through intermediate states $\mathbf{D}_1, \ldots, \mathbf{D}_{n-1}$."

Observe that executional entailment is defined over paths, not over arbitrary m-paths. This is because executional entailment describes the behavior of a complete system of transactions, which behaves like an atomic transaction since its execution is not interleaved with anything else. Intuitively, a more general m-path with "gaps" represents the execution of an incomplete system of transactions, since the gaps must be "filled in" by the execution of other, unspecified transactions.

Normally, users issuing transactions know only the initial database state $\mathbf{D}_0$, so defining transaction execution via (1) is not always appropriate. To account for this situation, the version of entailment in (2) allows us to omit the intermediate and the final database states. Informally, Statement (2) says that transaction ϕ can execute successfully starting from database $\mathbf{D}_0$. Formally, this statement is read as follows: "Under the transaction base $\mathbf{P}$, the transaction ϕ succeeds from database $\mathbf{D}$." When the context is clear, we simply say that transaction ϕ *succeeds*. Likewise, when statement (2) is not true, we say that transaction ϕ *fails*. In Section 4, we present an inference system that actually *finds* a database sequence, $\mathbf{D}_1, \ldots, \mathbf{D}_n$, that satisfies Statement (1) whenever a transaction succeeds.

4 Proof Theory for Concurrent Horn Transactions

Like classical logic, $\mathcal{CTR}$ has a "Horn" version, which has both a procedural and a declarative semantics. It is this property that allows a user to *program* transactions within the logic. This section defines the Horn subset of $\mathcal{CTR}$ and develops an SLD-style proof theory that is both sound and complete. Unlike classical logic programming, the proof theory presented here computes new database states *as well as* query answers. In this sense, it is similar to the proof theory of sequential Transaction Logic. However, the latter proof theory accounts only for sequential processes, while the proof theory developed here accounts for concurrent and atomic processes as well. This leads to the notion of *hot components*, defined as those subprocesses that are ready to execute.

In the Horn fragment of $\mathcal{CTR}$, database states are represented by a *Horn oracle*, as defined in Section 2.2. In addition, the transaction base $\mathbf{P}$ must satisfy certain conditions. The first condition is based on the idea of *concurrent serial goal*, defined below, which generalizes the notion of conjunctive query in classical database theory. We say that $b \leftarrow \phi$ is a *concurrent Horn rule* if b is an atomic formula and ϕ is a concurrent serial goal. All examples in this paper use concurrent Horn rules. Finally, we say that the combination of a transaction base $\mathbf{P}$ and a Horn data oracle $\mathcal{O}^d(\mathbf{D})$ is *concurrent Horn* if $\mathcal{P}$ is a set of concurrent Horn rules satisfying the following *independence condition*: `For every database state` $\mathbf{D}$`, predicate symbols occurring in rule-heads in` $\mathbf{P}$ `do not occur in rule-bodies in` $\mathcal{O}^d(\mathbf{D})$. Intuitively, the independence condition means that the database does not define predicates in terms of transactions. Thus, the rule $a \leftarrow b$ *cannot* be in the database if the rule $b \leftarrow c$ is in the transaction base. Observe that all relational databases satisfy the independence condition, since the data oracle returns only a set of atoms.

Definition 4.1 (Concurrent Serial Goal) A *concurrent serial goal* is any formula of the form:

- An atomic formula; or
- $\phi_1 \otimes \ldots \otimes \phi_k$, where each ϕ_i is a concurrent serial goal, and $k \geq 0$; or
- $\phi_1 \mid \ldots \mid \phi_k$, where each ϕ_i is a concurrent serial goal, and $k \geq 0$; or
- $\odot\, \phi$, where ϕ is a concurrent serial goal. □

4.1 Inference System $\Im^C$

This section develops an inference system for verifying that $\mathbf{P}, \mathbf{D}_0 \text{---} \models \psi$. Informally, this statement says that transaction ψ can successfully execute starting from state $\mathbf{D}_0$. The inference succeeds if and only if it finds an execution path for the transaction ψ, that is, a sequence of databases $\mathbf{D}_1, \ldots, \mathbf{D}_n$ such that $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \psi$. We refer to the inference system of this section as $\Im^C$.

	Formula	Hot Components
	$a \otimes b \otimes c$	$\{a\}$
long	$a \mid b \mid \odot(c \otimes d) \otimes e$	$\{a, b, \odot(c \otimes d)\}$
	$(a \mid b) \otimes (c \mid d)$	$\{a, b\}$
	$(a \otimes b) \mid (c \otimes d)$	$\{a, c\}$

Figure 1: Some formulas and their hot components.

The inference rules of $\Im^C$ focus on the left end of a transaction. Intuitively, this corresponds to the left-to-right execution of serial transactions. For instance, in the transaction $a_1 \otimes a_2 \otimes \ldots \otimes a_k$, the subtransaction a_1 executes first, then a_2, then a_3, and so on until a_k executes. Because of concurrency, there may be many subtransactions ready for execution at any given time. For instance, in the transaction $a_1 \mid a_2 \mid a_3$, any of the subtransactions a_1, a_2, a_3 can start executing first. The next definition formalizes this idea. It defines the set of subformulas that are ready for execution, which we refer to as "hot" components. Figure 1 illustrates the idea.

Definition 4.2 (Hot Components) Let ϕ be a concurrent serial goal. Its set of *hot components*, written $hot(\phi)$, is defined recursively as follows:

- $hot(\,(\,)\,) \;=\; \{\,\}$, where $(\,)$ is the empty goal;
- $hot(b) \;=\; \{b\}$, if b is an atomic formula;
- $hot(\psi_1 \otimes \ldots \otimes \psi_k) \;=\; hot(\psi_1)$;
- $hot(\psi_1 \mid \ldots \mid \psi_k) \;=\; hot(\psi_1) \cup \ldots \cup hot(\psi_k)$;
- $hot(\odot\,\psi) \;=\; \{\odot\,\psi\}$.

□

Just as databases and logic programs assume that queries are existentially quantified, we shall assume that processes are existentially quantified. We thus introduce the notion of an *existential* goal, which is a formula of the form $\exists \overline{X}\,\phi$, where ϕ is a concurrent serial goal. For convenience, we shall often drop the variable list $\overline{X}$ and simply write $(\exists)\,\phi$.

Definition 4.3 (Inference in $\Im^C$) If $\mathbf{P}$ is a concurrent Horn transaction base, then $\Im^C$ is the following system of axioms and inference rules, where $\mathbf{D}$ is any legal database state.

Axioms: $\mathbf{P}, \mathbf{D}\text{---} \vdash (\,)$, for any $\mathbf{D}$.

Inference Rules: In Rules 1–4 below, σ is a substitution, ψ and ψ' are concurrent serial conjunctions, and a is an atomic formula in $hot(\psi)$.

1. *Applying transaction definitions:* Let $b \leftarrow \beta$ be a rule in $\mathbf{P}$, and assume that its variables have been renamed so that none are shared with ψ. If a and b unify with mgu σ then

   $$\frac{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\,\psi'\,\sigma}{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\,\psi} \quad \text{where } \psi' \text{ is obtained from } \psi \text{ by replacing a hot occurrence of } a \text{ by } \beta.$$

   For instance, if $\psi = c \mid a \mid d$ then $\psi' = c \mid \beta \mid d$.

2. *Querying the database:* If $\mathcal{O}^d(\mathbf{D}) \models^c (\exists)a\sigma$, and $a\sigma$ and $\psi'\sigma$ share no variables, then

   $$\frac{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\,\psi'\,\sigma}{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\,\psi} \quad \text{where } \psi' \text{ is obtained from } \psi \text{ by deleting a hot occurrence of } a.$$

   For instance, if $\psi = c \mid a \mid d$ then $\psi' = c \mid d$.

3. *Executing elementary updates:* If $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists) a\sigma$ and $a\sigma$ and $\psi'\sigma$ share no variables, then

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\, \psi'\, \sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, \psi} \quad \text{where } \psi' \text{ is obtained from } \psi \text{ by deleting a hot occurrence of } a.$$

For instance, if $\psi = c \mid a \mid d$ then $\psi' = c \mid d$.

4. *Executing atomic transactions:* If $\odot\, \alpha$ is a hot component in ψ then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, (\alpha \otimes \psi')}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \psi} \quad \text{where } \psi' \text{ is obtained from } \psi \text{ by deleting a hot occurrence of } \odot\, \alpha.$$

For instance, if $\psi = c \mid (\odot\, \alpha) \mid d$ then $\psi' = (c \mid d)$. □

This system manipulates expressions of the form $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \psi$, called *sequents*. The informal meaning of such a sequent is that the transaction $(\exists)\, \psi$ can *succeed* from $\mathbf{D}$, *i.e.*, it can execute on a path starting at database $\mathbf{D}$. Each inference rule consists of two sequents, one above the other, and has the following interpretation: If the upper sequent can be inferred, then the lower sequent can also be inferred. As in classical resolution, any instance of an answer-substitution is a valid answer to a query.

To understand inference system $\Im^C$, first note that the axioms describe the empty transaction, "( )". By definition, this transaction does nothing and always succeeds. That is, if the user issues the command ?– ( ), then the database simply remains in its current state.[5] The axioms formalize this behavior. The four inference rules describe more complex transactions. To understand these rules, it is convenient to temporarily ignore the unifier, σ, and to assume that a is identical to b. With this in mind, we can interpret the rules as follows:

1. Inference rule 1 deals with defined transactions. It assumes that b is defined by β, and that β is a subtransaction of ψ'. The rule says that if transaction ψ' succeeds from database $\mathbf{D}$, then so does ψ. Intuitively, this rule replaces a subroutine definition, β, by its calling sequence, b.

2. Inference rule 2 deals with tests (*i.e.*, queries) that do not cause state changes. It assumes that test b is true at state $\mathbf{D}$. The rule says that if transaction ψ' succeeds from $\mathbf{D}$, then so does ψ. Intuitively, b is a pre-condition that is satisfied by $\mathbf{D}$, so it can be attached to the front of ψ'.

3. Inference rule 3 deals with elementary updates. It assumes that update b transforms the database from state $\mathbf{D}_1$ to state $\mathbf{D}_2$. The rule says that if transaction ψ' succeeds from $\mathbf{D}_2$, then transaction ψ succeeds from $\mathbf{D}_1$. Intuitively, b is attached to the front of ψ', so that the resulting transaction starts from $\mathbf{D}_1$ instead of $\mathbf{D}_2$.

4. Inference rule 4 deals with atomic subtransactions. It assumes that subtransaction α is ready to execute, and that it must execute atomically. To achieve atomicity, the rule extracts α from the main transaction, ψ; then it executes α to completion; and then it executes the rest of transaction ψ, which we have denoted ψ'. The formula $\alpha \otimes \psi'$ represents the execution of α followed by the execution of ψ'. Here α executes atomically since it is not interleaved with anything.

Example 4.4 (Deduction) Suppose the data oracle is relational and the transition oracle inserts propositional atoms into the database. Suppose also that the transaction base $\mathbf{P}$ contains the following two rules:

$$p \leftarrow c.ins \otimes d.ins \qquad\qquad q \leftarrow e.ins \otimes f.ins$$

Then the formula $p \mid q$ represents a transaction in which the atoms c, d, e, f are inserted into the database, where c is inserted before d, and e before f. This transaction can be executed in several ways. Figure 2 illustrates one possibility. It shows a derivation of the sequent $\mathbf{P}, \{\} \text{---} \vdash p \mid q$. Each sequent in the table is derived from the one below by an inference rule. The deduction succeeds because the bottom-most sequent is an axiom. When carried out top-down, the deduction corresponds to an execution of the transaction $p \mid q$ in which atoms are inserted into the empty database in the order c, e, f, d. □

[5] Model theoretically, "( )" is a transaction that is true on every m-path of the form $\langle \mathbf{D} \rangle$, *i.e.*, on any m-path corresponding to a single database [9, 6].

	Sequent	Inf. Rule	Hot Components
	$P, \{\} \text{---} \vdash p \mid q$	**1**	$\{p,\ q\}$
	$P, \{\} \text{---} \vdash (c.ins \otimes d.ins) \mid q$	**1**	$\{c.ins,\ q\}$
long	$P, \{\} \text{---} \vdash (c.ins \otimes d.ins) \mid (e.ins \otimes f.ins)$	**3**	$\{c.ins,\ e.ins\}$
	$P, \{c\} \text{---} \vdash (d.ins) \mid (e.ins \otimes f.ins)$	**3**	$\{d.ins,\ e.ins\}$
	$P, \{c, e\} \text{---} \vdash (d.ins) \mid (f.ins)$	**3**	$\{d.ins,\ f.ins\}$
	$P, \{c, e, f\} \text{---} \vdash (d.ins)$	**3**	$\{d.ins\}$
	$P, \{c, d, e, f\} \text{---} \vdash ()$	axiom	$\{\}$

Figure 2: A Deduction in System $\Im^C$.

Theorem 4.5 (Soundness and Completeness of $\Im^C$) *If* $\mathbf{P}$ *is a concurrent Horn transaction base, and* ψ *is a concurrent serial goal, then the executional entailment* $\mathbf{P}, \mathbf{D} \text{---} \models (\exists)\,\psi$ *holds if and only if there is a deduction in* $\Im^C$ *of the sequent* $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,\psi$*.*

Having developed the inference system, we must remind ourselves that our original goal was not so much proving statements of the form $\mathbf{P}, \mathbf{D} \text{---} \models (\exists)\,\phi$, but rather of the form $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models (\exists)\,\phi$, where $\mathbf{D}_0$ is the database state at the time the user issues the transaction $?-\ (\exists)\phi$. Note that the intermediate states, $\mathbf{D}_1, \ldots, \mathbf{D}_{n-1}$, and the final state, $\mathbf{D}_n$, are *unknown* at this time. An important task for the inference system is to compute these states. A general notion of deduction is not tight enough to do this conveniently, since general deduction may record the execution of many unrelated transactions, mixed up in a haphazard way. Since we are interested in the execution of a particular transaction, we introduce the more specialized notion of *executional deduction*, which—without sacrificing completeness—defines a narrower range of deductions. The notion of executional deduction needed here is exactly the same as that developed for sequential Transaction Logic in [9, 6]. It also leads naturally to a proof-theoretic notion of execution path. It remains only to show that executional deduction in concurrent Transaction Logic is sound and complete for the model theory of Section 3. This is established by the following theorem.

Theorem 4.6 (Executional Soundness and Completeness of $\Im^C$) *Let* $\mathbf{P}$ *be a concurrent Horn transaction base, and let* ϕ *be a concurrent serial goal. There is an executional deduction of* $(\exists)\,\phi$ *whose execution path is* $\mathbf{D}_0\ \mathbf{D}_1\ \ldots\ \mathbf{D}_n$ *if and only if the following entailment is true:*

$$\mathbf{P}, \mathbf{D}_0\ \mathbf{D}_1\ \ldots\ \mathbf{D}_n \models (\exists)\,\phi$$

5 Expressive Power

The ability to express the *send* and *receive* primitives using the modal operator $\odot$ provides for general communication and synchronization among $\mathcal{CTR}$ processes. A process can send a message and either choose to synchronize with the receiver immediately or go on doing its business, possibly synchronizing later. In other words, both synchronous and asynchronous communication are supported. To synchronize with another process, a process calls $receive(Ch, Msg)$, where Ch and Msg are (possibly non-ground) terms, as illustrated in Example 2.2. Such a process cannot advance beyond this call until a channel that unifies with Ch has a message that unifies with Msg. In that sense, $\mathcal{CTR}$ supports pattern-directed communication. Patterns (which in our case are function terms representing channels) can be arranged in various ways to achieve point-to-point, broadcast, blackboard, and other forms of communication, with bounded or unbounded buffers.

Dynamic Communication Topology: As in the π-calculus [26], $\mathcal{CTR}$ processes can send channel names to each other, through which they can communicate later, thereby changing the communication topology. To illustrate, the program below simulates a simple server that responds to client requests. Intuitively, the predicate $client(Id)$ represents a client process with identifier Id. The predicate $server(N)$ represents a server process that can respond to N client requests before expiring. The client and server processes can be started in any number of ways. For example, to create three clients and a server that can respond to three requests, the system administrator could issue the following command:

$$?-\ server(3)\ \mid\ client(c1)\ \mid\ client(c2)\ \mid\ client(c3).$$

where the constants $c1, c2, c3$ are client identifiers. The immediate effect of this command is to create four concurrent processes: one server process, and three client processes. As the clients send requests to the server, additional concurrent processes will be created by the server.

$$\begin{array}{l}
server(0) \longleftarrow \\
server(N) \longleftarrow receive(servConnect, Ch) \otimes [servClient(Ch) \mid server(N-1)] \\
servClient(Ch) \longleftarrow receive(Ch, Request) \otimes doIt(Request, Result) \otimes send(Ch, Result) \\
\\
client(Id) \longleftarrow newChannel(Ch) \otimes \cdots \otimes send(servConnect, Ch) \otimes clientDoWork(Id, Ch) \\
clientDoWork(Id, Ch) \longleftarrow send(Ch, Req) \otimes \cdots \otimes receive(Ch, Result) \otimes \cdots \otimes delChannel(Ch)
\end{array}$$

Here, $servConnect$ is the name of a public communication channel. The predicate $newChannel(Ch)$ is an atomic process that creates a private channel and returns its name, Ch. The predicate $delChannel(Ch)$ is an atomic process that destroys channel Ch. $newChannel$ and $delChannel$ are easily defined via the transition oracle or the transaction base [9].

A client process $client(Id)$ first generates a new channel, Ch. It then connects to the server via the public channel, passing it the new channel name. The server then spawns a subprocess $servClient(Ch)$ to deal with the client. Future communication between the client and this server subprocess takes place along the new channel, Ch. This communication has a simple form: the client sends a request Req to the server subprocess, which carries out the request, and sends the result back to the client. Finally, the client destroys its private channel and terminates.

Passing Process Names: In addition to sending channel names between processes, in $\mathcal{CTR}$, one can send process names between processes. One way to do this is to give each process an id encoded as a function term, like the client id's used above. Process names can then be sent and received like other messages, and executed like other processes. Another way to do this is to extend $\mathcal{CTR}$ in the direction of HiLog [12], a well-known higher-order extension of classical logic.

In a nutshell, HiLog allows variables over functions, predicate symbols, and even over atomic formulas. Moreover, the distinction between atomic formulas and terms is completely erased, and arbitrary (even non-ground) terms can serve as predicate names in HiLog, which provides a convenient way of defining generic logical procedures [12]. For instance, $A(X,Y)(Z)(V,W)$ is a well-formed term in HiLog. As a concrete example, the rules below define a predicate $fact(N)(A)$ that computes the factorial function. The definition is pretty much classical except that $fact(N)$ is treated as a parameterized predicate, with the parameter serving as input. Thus, $fact(20)(A)$ means compute 20! and return the result in A. The reason for using HiLog here is that it lets us transmit and execute processes without knowing their precise calling sequence.

To illustrate these ideas, consider the two processes $proc1$ and $proc2$ defined below. To see how they work, suppose the user issues the command $?-\ proc1(ch1, fact(20), A) \mid proc2(ch1)$. First, the process $proc1$ sends the term $fact(20)$ out along channel $ch1$. Process $proc2$ then receives $fact(20)$, interprets it as a process, and executes it. Finally, $proc2$ sends the answer out along channel $ch1$, whence $proc1$ receives the answer and returns it to the user.

$$\begin{array}{l}
proc1(Ch, Process, Ans) \longleftarrow send(Ch, Process) \otimes receive(Ch, Ans) \\
proc2(Ch) \longleftarrow receive(Ch, Process) \otimes Process(Ans) \otimes send(Ch, Ans) \\
\\
fact(0)(1) \longleftarrow \\
fact(N)(N * A) \longleftarrow fact(N-1)(A)
\end{array} \tag{3}$$

Guarded Clauses: Finally, a remark on the relationship between $\mathcal{CTR}$ and concurrent logic programming (CLP) languages [32] is in order. Since the semantics of these languages has eluded capture in logic, we will not pretend that $\mathcal{CTR}$ can simulate these languages faithfully. However, some analogies can be made. For instance, CLP languages commonly use the idea of rules with *guarded bodies*, which can have the following form:

$$head \longleftarrow g_1, \ldots, g_n \mid a_1, \ldots, a_m \tag{4}$$

When *head* unifies with the goal, the *guards* $g_1, \ldots, g_n$ are evaluated. If they all succeed, then the subgoals $a_1, \ldots, a_n$ are executed concurrently. An important point here is that if one of the subgoals a_i fails, then *head* also fails, even though another clause for *head*—had it been chosen instead of (4)—could have succeeded. The $\mathcal{CTR}$ analogue of (4) is:

$$head \longleftarrow (g_1 \mid \ldots \mid g_n) \otimes (a_1 \mid \ldots \mid a_m)$$

A crucial difference with guarded rules is that $\mathcal{CTR}$ does not rely on committed choice, which is so inherent to CLP languages. Thus, if some a_i fails, then another clause for *head* will be tried.

The importance of committed choice in CLP languages is that it helps control non-determinism, leading to greater efficiency. However, committed choice has been criticized for its distinctly non-logical semantics that may cause a system to commit to a clause that would later fail, even though another clause would have succeeded. In contrast, $\mathcal{CTR}$ offers a different paradigm, one with clean, logical model and proof theories and several ways of controlling non-determinism via synchronization. Non-determinism has entirely different nature here ("don't know" rather than "don't care" [32]), and it is deeply rooted in the model theory.

6 Conclusions and Discussion

We presented Concurrent Transaction Logic ($\mathcal{CTR}$), a logic that can declaratively specify and procedurally execute concurrent communicating database processes involving queries, updates, or combinations of both. It is thus a high-level deductive language for programming database applications, a language with a completely logical semantics. The semantics emphasizes the *combination* of elementary database operations into complex processes. The elementary operations themselves are not built into the semantics of $\mathcal{CTR}$, but rather play the role of a parameter to the logic. In this paper, we focused on the concurrent Horn subset of $\mathcal{CTR}$ and developed its model theory and a sound and complete proof theory.

$\mathcal{CTR}$ is not yet-another-logic unrelated and isolated from other extensions. It is an idea that is orthogonal to several other recent proposals, such as F-logic [17] (structural object-orientation), HiLog [12] (higher-order logic programming), and Annotated Logic [3, 18, 19] (reasoning with uncertainty), and it can be easily integrated with them to endow these static formalisms with the ability to capture database dynamics in a clean, logical fashion (cf., *e.g.*, [16]).

The communication paradigm within $\mathcal{CTR}$ is inspired by the π-calculus [26, 27]. However, $\mathcal{CTR}$ is a *programming* logic, while π-calculus is an algebra used for *specifying* and *verifying* finite-state concurrent systems (which databases and logic programs are not). Although there is growing interest in designing programming languages based on π-calculus (e.g., PICT [30]), the application domain of such languages seems very different from $\mathcal{CTR}$. These languages have a functional flavor, and their ability to express data-driven non-determinism is limited. Once a process is committed to a certain execution path, there is no possibility for failure. In general, it is hard to specify pre-conditions and post-conditions in π-calculus, and search-related problems, such as those arising in databases and AI are difficult to program. Compared to π-calculus, concurrency in $\mathcal{CTR}$ is more flexible in some respects, and more limited in others. For instance, $\mathcal{CTR}$ supports both synchronous and asynchronous communication, while the π-calculus requires a hand-shake. Like the π-calculus, $\mathcal{CTR}$ can send and receive messages and channels, and it can reconfigure the communication topology dynamically. To a large extent, $\mathcal{CTR}$ processes can create *private communication channels*, á la π-calculus. With a simple extension in the direction of HiLog [12], $\mathcal{CTR}$ can pass transactions between processes for remote execution.

In a recent work, Miller [25] has shown that most of the π-calculus can be encoded in Linear Logic [13]. The reduction process of the π-calculus is simulated via the proof theory of Linear Logic. However, the semantics of Linear Logic does not give direct meaning to execution and communication, unlike $\mathcal{CTR}$. Also, the programming paradigm that might arise from such an encoding seems to imitate the behavior of

π-calculus, which is quite different from the programming style of $\mathcal{CTR}$, which draws on deductive databases and logic programming.

Linear Objects is another concurrent formalism based on Linear Logic [1]. Processes are represented via atomic formulas and their execution corresponds to branches in the proof tree. The semantics is provided via a mapping into Linear Logic, although, strictly speaking, this does not yet provide a model-theoretic account of concurrency. Furthermore, although processes can communicate, there does not seem to be an obvious way for one process to change the database and for others to query the changes (or to have different views of the changes). There is also no programming language for specifying how processes evolve. Instead, one must list all the operations performed by a process (and its subprocesses) before execution begins.

Another related formalism, *rewriting logic* [22], is a general framework for specifying executable concurrent systems. In fact, it is so general that classical logic, Linear Logic, the π-calculus, and $\mathcal{CTR}$ can all be encoded as particular theories within this logic. Rewriting logic serves two purposes: as a general framework within which to compare different concurrent formalisms, and as a workbench for designing languages for programming concurrent systems. In the first capacity, we believe that an embedding of $\mathcal{CTR}$ into rewriting logic may prove useful for clarifying the relationship between $\mathcal{CTR}$ and other concurrent formalisms. In the second capacity, rewriting logic has been used to design languages for object-oriented programming [23, 24], and equational programming [23]. However, the programming paradigm of $\mathcal{CTR}$ is fundamentally different from these other languages.

Some of the earliest attempts at adding concurrency to logic programming were PARLOG, Concurrent Prolog, GHC, and related languages (see [32] for a survey). We discussed possible connections between $\mathcal{CTR}$ and concurrent logic programming earlier in this paper. Although concurrent logic programming languages lack model-theoretic semantics, the programming paradigm promoted by these languages is related to rewriting logic and linear objects in the sense that once the flow of control commits to a certain action, this action never backtracks; this style is quite unlike the one promoted by $\mathcal{CTR}$.

Finally, a comparison of $\mathcal{CTR}$ with Concurrent Dynamic Logic (CDL) [29, 28] is in order. There are two versions of CDL. The version developed in [29] is very different from $\mathcal{CTR}$. It is modeled on the kind of concurrency found in alternating Turing machines, which does not allow for communication between concurrent processes. A deductive-database analogue of this version of CDL is Hypothetical Datalog [5], in which hypothetical databases represent the states of the various (non-communicating) concurrent processes in an alternating computation. The version of CDL developed in [28] does allow for communication, but only after adding considerable complexity to the semantics.

In both versions of CDL, the meaning and intent of dynamic formulas is fundamentally different from that of transaction formulas in $\mathcal{CTR}$. CDL was intended to *reason about* what is true during program execution, while $\mathcal{CTR}$ was designed to actually *execute* declaratively specified procedures. This difference in the intent is reflected in the syntax. For instance, CDL uses a separate alphabet to represent actions, and a set of modal operators to reason about them. Thus, unlike $\mathcal{CTR}$, CDL processes are not represented as propositions. In particular, processes are not logical formulas, but rather are terms used in constructing modal operators. One consequence is that only elementary actions have names. Composite actions cannot be named, and thus the logic *lacks a subroutine facility.* Another difference between $\mathcal{CTR}$ and CDL is in the nature of states. In $\mathcal{CTR}$, the concept of a state immediately leads to the idea of data and transition oracles. In contrast, CDL worries about internal states of executing programs, while the notion of a database state is entirely missing. It is also worth noting that these statements equally apply to another related logic, *Process Logic* [15], although we are not aware of any extensions of Process Logic that allow concurrency and communication.

Acknowledgments. Anthony Bonner's work was supported in part by an Operating Grant from the Natural Sciences and Engineering Research Council of Canada. Michael Kifer was supported in part by the NSF Grant IRI-9404629. We thank Mariano Consens for discussions on preliminary ideas for concurrency in Transaction Logic. We are also grateful to Jose Meseguer for his helpful comments.

References

[1] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(4):445–473, 1991.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases*. Addison Wesley, 1987.

[3] H.A. Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[4] A. Bonner, A. Shrufi, and S. Rozen. LabFlow-1: a database benchmark for high-throughput workflow management. In *Intl. Conference on Extending Database Technology*, number 1057 in Lecture Notes in Computer Science, pages 463–478, Avignon, France, March 25–29 1996. Springer-Verlag.

[5] A.J. Bonner. Hypothetical Datalog: Complexity and expressibility. *Theoretical Computer Science*, 76:3–51, 1990.

[6] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming*, pages 257–282, Budapest, Hungary, June 1993. MIT Press.

[7] A.J. Bonner and M. Kifer. Applications of transaction logic to knowledge representation. In *Proceedings of the International Conference on Temporal Logic*, number 827 in Lecture Notes in Artificial Inteligence, pages 67–81, Bonn, Germany, July 1994. Springer-Verlag.

[8] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[9] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.

[10] A.J. Bonner and M. Kifer. Concurrent Transaction Datalog: Complexity and expressibility. In preparation, 1996.

[11] A.J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In A. Ohori C. Beeri and D.E. Shasha, editors, *Proceedings of the International Workshop on Database Programming Languages*, Workshops in Computing, pages 309–337. Springer-Verlag, February 1994. Workshop held on Aug 30–Sept 1, 1993, New York City, NY.

[12] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[13] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[14] Nathan Goodman, Steve Rozen, and Lincoln Stein. Requirements for a deductive query language in the MapBase genome-mapping database. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, pages 259–278. Kluwer, 1994. `ftp://genome.wi.mit.edu/pub/papers/Y1994/requirements.ps`.

[15] D. Harel, D. Kozen, and R. Parikh. Process Logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, October 1982.

[16] M. Kifer. Deductive and object-oriented data languages: A quest for integration. In *Intl. Conference on Deductive and Object-Oriented Databases*, volume 1013 of *Lecture Notes in Computer Science*, pages 187–212, Singapore, December 1995. Springer-Verlag. Keynote address at the 3d Intl. Conference on Deductive and Object-Oriented databases.

[17] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, pages 741–843, July 1995.

[18] M. Kifer and E.L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, November 1992.

[19] M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12(4):335–368, April 1992.

[20] J.N. Maksym, A.J. Bonner, C.A. Dent, and G.L. Hemphill. Machine Analysis of Acoustical Signals. *Pattern Recognition*, 16(6):615–625, 1983. Also appears in *Proceedings of the Workshop on Issues in Acoustic Signal/Image Processing and Recognition*, San Miniato, Italy, August 5–9 1982.

[21] J.M. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969. Reprinted in *Readings in Artificial Intelligence*, 1981, Tioga Publ. Co.

[22] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[23] J. Meseguer. Multiparadigm logic programming. In *Algebraic and Logic Specifications*, number 632 in Lecture Notes in Computer Science, pages 158–200. Springer-Verlag, September 1992.

[24] J. Meseguer. A logic theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*. MIT Press, 1993.

[25] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In *Proceedings of the Workshop on Extensions to Logic Programming*, Lecture Notes in Computer Science, pages 242–265. Springer-Verlag, 1992.

[26] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[27] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, September 1992.

[28] D. Peleg. Communication in concurrent dynamic logic. *Journal of Computer and System Sciences*, 35(1):23–58, August 1987.

[29] D. Peleg. Concurrent-dynamic logic. *Journal of ACM*, 34(2):450 – 479, March 1987.

[30] B. Pierce. Programming in the π-calculus—An experiment in concurrent language design. PICT Version 3.4c tutorial. `ftp://ftp.dcs.ed.ac.uk/pub/bcp/pict.tar.Z`, March 1994.

[31] R. Reiter. Formalizing database evolution in the situation calculus. In *Conference on Fifth Generation Computer Systems*, 1992.

[32] E. Shapiro. A family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3), 1989.