Transaction Datalog: a Compositional Language for Transaction Programming

Anthony J. Bonner

University of Toronto, Department of Computer Science
Toronto, ON, Canada M5S 1A4
bonner@db.toronto.edu

Abstract

In the classical model of database transactions, large transactions cannot be built out of smaller ones. Instead, transactions are modelled as atomic and isolated units of work. This model has been widely successful in traditional database applications, in which transactions perform only a few simple operations on small amounts of simply-structured data. Unfortunately, this model is inappropriate for more complex applications in which transactions must be combined and coordinated to achieve a larger goal. Examples include CAD, office automation, collaborative work, manufacturing control, and workflow management. These applications require new transaction models, new methods of transaction management, and new transaction languages. This paper focuses on the latter issue: languages for specifying non-classical transactions, and combining them into complex processes. In particular, we develop *Transaction Datalog*, a deductive language that integrates queries, updates, and transaction composition in a simple logical framework. This integration extends the deductive-database paradigm with several new capabilities. For instance, Transaction Datalog supports all the properties of classical transactions, such as persistence, atomicity, isolation, abort and rollback. It also supports properties found in many new transaction models, such as subtransaction hierarchies, concurrency within individual transactions, cooperation between concurrent activities, a separation of atomicity and isolation, and fine-grained control over abort and rollback. These capabilities are all provided within a purely logical framework, including a natural model theory and a sound-and-complete proof theory. This paper outlines the problems of developing a compositional transaction language, illustrates our solution (Transaction Datalog) through a series of examples, and develops its formal semantics in terms of a logical inference system.

A short version of this paper appears in *Proceedings of the Sixth International Workshop on Database Programming Languages (DBPL)*, Estes Park, Colorado, August 18–20 1997, pages 303–322.

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

1 Introduction

Database transactions were originally modeled as atomic and isolated units of work, with no internal structure and no external connections [9]. This "classical" transaction model has been widely successful for applications like banking, airline reservations, and inventory control, where transactions perform only a few simple operations on small amounts of simply-structured data. Unfortunately, this model is inappropriate for more complex applications in which transactions must be combined and coordinated to achieve a larger goal. This need is typical of new database applications involving distributed systems, complex data structures, and cooperation between multiple users or multiple concurrent processes. Examples include CAD, office automation, collaborative work, manufacturing control, and workflow management. Such applications combine database transactions, application programs, and other activities into larger information systems and business processes [32, 40, 39, 10, 42, 34, 21, 43, 45]. These applications require new transaction models, new methods of transaction management, and new transaction languages [32, 40, 39, 29].

This paper focuses on the latter issue: languages for specifying non-classical transactions, and for combining them into complex processes. In particular, we argue that logic provides a natural basis for such languages. The main contribution is a new deductive language called *Transaction Datalog* (abbreviated $\mathcal{TD}$). $\mathcal{TD}$ has a natural model theory and a sound-and-complete proof theory, and it extends the paradigm of deductive databases with several new capabilities. For instance, in addition to declarative queries and views, $\mathcal{TD}$ provides (*i*) updates and nested transactions, (*ii*) composition of transaction programs, and (*iii*) concurrency and communication. In addition, it provides a smooth integration of procedural and declarative programming, and in the absence of updates, it reduces to classical Datalog.

Transaction Datalog is derived from a general logic of state change called *Transaction Logic* [14, 16, 13, 19, 15]. Transaction Logic allows users to express properties of transaction programs, and to reason about them [18]. For instance, one can reason about when a program will commit or abort, and about whether a program preserves integrity constraints. In addition, like classical logic, Transaction Logic has a "Horn" fragment with both a procedural and declarative semantics. This fragment provides a logic programming language in which users can specify and execute database transactions. Transaction Datalog is derived from this Horn fragment by restricting it to relational databases and to rules without function symbols (*i.e.*, just as classical Datalog is derived from classical Horn logic). Transaction Datalog thus inherits the semantics of the full logic, which has been published elsewhere [14, 16, 13, 19]. However, because Transaction Datalog is a specialized system, it has a specialized semantics, which is simpler than the more-general semantics of the full logic. This paper develops the simplified semantics in terms of a logical inference system. The paper also illustrates the properties of $\mathcal{TD}$ through a series of examples. The examples show how logical formulas in $\mathcal{TD}$ can be interpreted both procedurally and declaratively. They also show how the logical structure of $\mathcal{TD}$ naturally captures many basic properties of non-classical transactions.

Related papers on Transaction Logic, a prototype implementation, and the results of benchmark tests are available at the Transaction Logic web-page: `http://www.cs.toronto.edu/~bonner/transaction-logic.html`

1.1 Background

The limitations of the classical transaction model are well-documented in the literature (*e.g.*, [39, 55, 40, 32]). One important limitation is that this model does not support the composition of transaction programs. For instance, database transactions are usually defined by embedding SQL commands within a host programming language. Unfortunately, there are severe restrictions on the ability of embedded SQL to combine simple transaction programs into larger ones, regardless of the host language. These restrictions greatly hinder the modular development of large transaction programs. This problem is not limited to embedded SQL, but is shared by almost all application programming languages for commercial database systems, since these systems are based on the classical transaction model.

The first attempt to address this problem lead to the nested transaction model, in which a transaction can be composed of subtransactions [55, 40]. As a simple example, suppose we have a transaction program for withdrawing money from a bank account, and another for depositing money. We would like to compose these two programs into a third program for transferring money from one account to another, and we would like this third program to execute as a transaction, *i.e.*, as an atomic and isolated unit of work. Of course, we could write a money-transfer program from scratch in embedded SQL, but that is not the point. The point is to reuse and combine existing transaction programs. In particular, we would like to execute the withdraw and deposit programs concurrently; and if one fails, we would like them both to abort, and their effects on the database to be undone. This requirement poses several serious problems for the classical transaction model, and for transaction management systems based on it. First, the withdraw and deposit transactions are not independent. In particular, the failure of one implies the failure of the other, even if the other has successfully completed its execution and has committed. Second, we now need serializability *within* transactions, not just between them. In particular, the withdraw and deposit transactions must be executed serializably within the transfer transaction. Third, composite transactions can now behave like atomic and isolated units of work. In particular, the transfer program must execute to completion or not at all (atomicity), and its execution with other transactions must be serializable (isolation). These requirements are not supported by most commercial products. In particular, they cannot be met by having application programmers specify transactions in a conventional programming language on top of a conventional DBMS (*e.g.*, by using SQL embedded in C, or even in concurrent C).

As another example, consider the following abstract process, taken from [72]:

> *Run Transaction T1. Then execute transactions T2, T3, and T4 in parallel. Immediately after their successful completion, start T5. But, if one of T2, T3, or T4 fails, then abort the other two. In this case, the effects of T1 have to be cancelled as well.*

This process is a composition of five transactions, *T1–T5*. As in the previous example, the transactions are not independent, and the failure of one can require that others be undone, even if they have already completed and committed. This dependence conflicts with the classical transaction model, which assumes that separate transactions are unrelated units of work. Such dependencies are typical of many new database applications, in which transactions participate in a complex web of relations. These new applications require the development of new transaction models. This need has been eloquently expressed by Jim Gray [32, page xvii]:

> *The transaction concept has emerged as the key structuring technique for distributed data and distributed computations. Originally developed and applied to database applications, the transaction model is now being used in new application areas ranging from process control to cooperative work. Not surprisingly, these more sophisticated applications require a refined and generalized transaction model. The concept must be made recursive, it must deal with concurrency within a transaction, it must relax the strict isolation among transactions, and it must deal more gracefully with failures.*

Many new transaction models have been proposed in the literature. Nested Transactions were the first [55, 40]. More recent models include Sagas [37], ConTracts [72], Flex Transactions [33], Cooperative Transactions [59], Multi-Level Transactions and Open Nested Transactions [75], among others [32]. Much of the research on these models emphasizes transaction *management*. The focus has therefore been on systems issues such as concurrency control and recovery, locking protocols, distributed commit and abort, fault tolerance, scheduling, implementation and performance.

In addition to new methods of transaction management, new transaction *languages* are also needed [29]. These languages must deal *both* with conventional programming issues *and* with transactional issues. For instance, they must allow transaction programs to be combined sequentially, concurrently, and hierarchically. In addition, they must deal with persistent data and with transaction abort, rollback, atomicity, and isolation. Moreover, they must deal with these issues both for elementary transactions and for composite transactions. For example, suppose that a number of small transaction programs are combined into a larger program. Numerous questions about the larger program immediately arise. Does it execute as a transaction? Does it execute atomically? Does it execute in isolation? If some of the small transaction programs abort, does the larger program abort as well? If so, does it abort completely or partially? What effect does this have on the database? What effect does this have on the program's execution state? These questions must be addressed by any language that supports the composition of transaction programs.

The database systems community has begun to address these questions. For instance, some transaction programming languages offer save points, which support a limited form of nested transactions and partial rollback. In addition, *Transactional-C* is a commercial programming language for the Encina TP monitor, which provides full support for nested transactions [68]. Likewise, a number of research projects have developed programming languages for nested transactions [48, 35] and other non-classical transaction models [72, 56, 71].

Unfortunately, although some programming languages have been implemented and others have been proposed, their theoretical foundations are incomplete. In general, the theory of non-classical transactions has focussed on transaction management (*e.g.*, [8, 49, 74, 26, 6]), not on transaction languages. For instance, there has been no attempt to integrate relational algebra and relational updates into a language for transaction composition. Likewise, issues such as declarative semantics, data complexity, and transaction expressibility have been completely ignored. These issues have been studied extensively in the context of classical transactions and database queries (*e.g.*, [23, 24, 70, 2, 3, 1]). The challenge is to extend this theory to non-classical transactions. This paper takes a first step.

1.2 Transaction Datalog

In this paper, we propose a logic-based approach to the problems of specifying non-classical transactions. In particular, we develop *Transaction Datalog* (or $\mathcal{TD}$), a deductive database language for specifying transactions and combining simple transactions into complex ones. Like classical Datalog, $\mathcal{TD}$ has *both* a declarative semantics *and* an equivalent operational semantics. The declarative semantics includes a logical model theory and sound-and-complete inference system. The operational semantics includes an SLD-style proof procedure in the logic-programming tradition [14, 16, 13, 19, 15]. This procedure executes transactions and updates the database as it proves theorems. Transaction Datalog is a minimal language based on a few simple operations. However, these operations lead directly to a wide range of transactional and programming capabilities. For instance, $\mathcal{TD}$ supports all the properties of classical transactions, such as persistence, atomicity, isolation, abort and rollback. It also supports many properties found in non-classical transaction models, such as subtransaction hierarchies, concurrency within individual transactions, cooperation between concurrent activities, a separation of atomicity and isolation, and fine-grained control over abort and rollback. Moreover, these features are seamlessly integrated with the traditional features of classical deductive databases, namely declarative queries and views. In fact, in the absence of updates, Transaction Datalog reduces to classical Datalog. It therefore represents a conservative extension of the deductive-database paradigm.

This extension is possible because, unlike ordinary programs, transactions either commit (succeed) or abort (fail). We can therefore associate a truth value with each execution of a transaction program, where *true* corresponds to commit, and *false* corresponds to abort. Based on this idea, we develop a logical calculus for combining transaction programs, including connectives for sequential and concurrent composition, and a modality for specifying isolation. All formulas in the calculus represent transaction programs. In the declarative semantics, a formula specifies a program's legal execution traces (Section 3). In the operational semantics, the formula is evaluated as the program executes; if at any point the formula evaluates to false, then the execution is aborted and the database is rolled back to an earlier state (Section 2). In $\mathcal{TD}$, calculus formulas are used as rule bodies. In this way, users can define named procedures (such as views and subroutines), exactly as in deductive databases and logic programming.

Like classical Datalog, Transaction Datalog can be embellished with negation-as-failure. When this is done, $\mathcal{TD}$ can simulate a number of different transaction models. For simplicity, though, this paper focuses on the negation-free version of the language, which is well-suited to specifying nested transactions [55, 40]. In this model, a transaction may be decomposed into subtransactions. These subtransactions may execute serially or concurrently, and their effects are undone if the parent transaction aborts, even if the subtransactions have already committed. "Nested transactions provide a powerful mechanism for fine-tuning the scope of rollback in applications with a complex structure" [40]. Moreover, "there is a strong relationship between the concept of modularization in software engineering and the nested transaction mechanism" [40]. These properties make nested transactions ideal for distributed applications, object-oriented databases, and layered software systems [75, 4, 5, 55]. Numerous examples in this paper deal with nested transactions.

In addition to transactional features, Transaction Datalog provides all the functionality of a declarative query language and a procedural programming language, seamlessly integrated. To see this, it is instructive to compare and contrast Transaction Datalog with embedded SQL (*e.g.*, SQL embedded in C). Like embedded SQL, Transaction Datalog is a database programming language for defining queries, updates and transactions. Both languages integrate programming constructs with database access. However, unlike embedded SQL, Transaction Datalog is a single, unified formalism, not an amalgamation of two formalisms (SQL and C). In particular, Transaction Datalog does not make a sharp distinction between declarative programming (SQL) and procedural programming (C). In fact, because it has a logic-programming foundation, Transaction Datalog provides a seamless integration of procedural and declarative programming styles. For instance, users can write classical Datalog queries, and they can write sequential and concurrent algorithms, and they can write programs that are neither procedural nor declarative, but somewhere in between. The result is that Transaction Datalog avoids many of the problems of embedded SQL, such as the infamous "impedance mismatch" problem [28, 7, 62]. Of course, Transaction Datalog can also *compose* transactions and define *nested* transactions, which goes well beyond the capabilities of embedded SQL.

2 Overview of Transaction Datalog

This section introduces Transaction Datalog informally through a series of simple examples. The examples show how logical formulas in $\mathcal{TD}$ can be interpreted procedurally and declaratively, and how they lead quickly to the basic properties of nested transactions. More-involved examples are given in Section 4.

As in any programming language, programs in Transaction Datalog are ultimately built from a set of elementary operations. In the case of *database* programming languages (like $\mathcal{TD}$), these operations are elementary database transactions. The precise set of elementary operations is somewhat arbitrary, and in this paper, four are provided. These operations are simple, they can be efficiently implemented, and they lead to expressive completeness [12]. They are also minimal, since removing any one of them causes a loss of expressive completeness [12]. To represent these four operations, we use four types of expression: q, $empty.r$, $ins.q$, $del.q$. The first two expressions are yes/no queries. Intuitively, q means "*Is atom q in the database*," and $empty.r$ means "*Is relation r empty.*" The other two expressions are updates. Intuitively, $ins.q$ means "*Insert atom q into the database*," and $del.q$ means "*Delete atom q from the database.*" These four elementary operations are transactions. The two updates are transactions that always succeed; and the two queries are transactions that succeed if they return "yes," and fail if they return "no." We shall see that the queries can be used as tests and conditions to force larger, composite transactions to fail. In the examples below, we adopt the Prolog convention that variables begin in upper case, and constants begin in lower case.

2.1 Sequential Transactions

To combine transaction programs sequentially, $\mathcal{TD}$ includes a logical connective called *serial conjunction*, denoted $\otimes$. Intuitively, if formulas ϕ_1 and ϕ_2 represent transaction programs, then the formula $\phi_1 \otimes \phi_2$ represents their sequential composition, that is, program ϕ_1 followed by program ϕ_2. Thus, the formula $del.q(a) \otimes ins.r(a)$ deletes the atom $q(a)$ from the database, and then inserts the atom $r(a)$. Formulas of the form $\phi_1 \otimes \phi_2 \otimes \cdots \otimes \phi_n$ are called *serial programs.*

To assign a name to a program, $\mathcal{TD}$ uses Horn-like rules. Intuitively, if p is an atomic formula, and ϕ is a program, then the rule $p \leftarrow \phi$ is a procedure definition, where p is the procedure name and ϕ is the procedure body. Thus, the formula $p(X) \leftarrow del.q(X) \otimes ins.r(X)$ defines $p(X)$ to be the program $del.q(X) \otimes ins.r(X)$. The variable X is a parameter of the procedure, and is bound to a constant symbol at run time. Rules may be recursive.

Example 2.1 (Financial Transactions: I) Suppose the balance of a bank account is given by the relation $balance(Acct, Amt)$. The rules below define four transaction programs: $change\_balance(Acct, Bal_1, Bal_2)$, to change the balance of account $Acct$ from Bal_1 to Bal_2; $withdraw(Amt, Acct)$, to withdraw an amount from an account; $deposit(Amt, Acct)$, to deposit an amount into an account; and $transfer(Amt, Acct_1, Acct_2)$, to transfer an amount from one account to another.

$$transfer(Amt, Acct_1, Acct_2) \leftarrow withdraw(Amt, Acct_1) \otimes deposit(Amt, Acct_2)$$

$$\begin{aligned} withdraw(Amt, Acct) \leftarrow& \\ & balance(Acct, Bal) \otimes Bal > Amt \otimes change\_balance(Acct, Bal, Bal - Amt) \end{aligned}$$

$$deposit(Amt, Acct) \leftarrow balance(Acct, Bal) \otimes change\_balance(Acct, Bal, Bal + Amt)$$

$$change\_balance(Acct, Bal_1, Bal_2) \leftarrow del.balance(Acct, Bal_1) \otimes ins.balance(Acct, Bal_2)$$

In each rule, the premises are evaluated from left to right. For instance, the first rule says: to transfer an amount, Amt, from $Acct_1$ to $Acct_2$, first withdraw Amt from $Acct_1$; and then, if the withdrawal succeeds, deposit Amt in $Acct_2$. 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; then, 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 $Bal > Amt$ 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. $\Box$

A transaction defined by serial conjunction succeeds if and only if each of its subtransactions succeed. More formally, the transaction $\phi_1 \otimes \phi_2$ succeeds if and only if both ϕ_1 and ϕ_2 succeed (which is why $\otimes$ is called serial *conjunction*). This implies that the failure of a subtransaction can cause the failure of its parent transaction. For instance, in Example 2.1, the *transfer* transaction fails if either of the subtransactions *withdraw* and *deposit* fail. Likewise, the *withdraw* transaction fails if the test $Bal > Amt$ fails. In the terminology of nested transactions, ϕ_1 and ϕ_2 are *vital* subtransactions of $\phi_1 \otimes \phi_2$, since both are crucial to its success.

Serial conjunction leads immediately to a basic property of nested transactions—*relative commit.* For instance, in the transaction $\phi_1 \otimes \phi_2$, if subtransaction ϕ_2 fails, then the whole transaction fails and must be undone. In particular, subtransaction ϕ_1 must be undone, even though it has already succeeded (and committed). Thus, subtransaction commits are not irrevocable, and can be undone if the parent transaction fails. The following is a more concrete illustration of this phenomenon.

Example 2.2 (Relative Commit) Consider a transaction involving two transfers, defined as follows:

$$transfer(fee, client, broker) \otimes transfer(cost, client, seller) \tag{1}$$

This transaction transfers a fee from a client to a broker, and then transfers a cost from the client to a seller. The transaction succeeds if and only if both transfers succeed. In a successful execution, the first transfer succeeds (and commits), and then the second transfer succeeds (and commits). However, suppose that the first transfer succeeds, and then the second transfer fails (due to lack of funds). In this case, the whole transaction fails, and is undone. In particular, even though the first transfer has already committed, its effects are undone, and the database is restored to its initial state. Thus, the commit of the first transfer was not absolute, but was relative to the overall transaction. In this way, the whole transaction (like the individual transfers) behaves like an atomic operation, which executes to completion or not at all. □

A transaction defined by a rule succeeds if the rule body succeeds. More formally, given the rule $p \leftarrow \phi$, then p succeeds if ϕ succeeds. This leads immediately to non-determinism. For instance, suppose we are given the rules $p \leftarrow \phi_1,\ p \leftarrow \phi_2, \ldots\ p \leftarrow \phi_n$. Then, p succeeds if ϕ_1 succeeds, and p succeeds if ϕ_2 succeeds, and p succeeds if ϕ_3 succeeds, etc. Thus, p succeeds if *some* ϕ_i succeeds. Intuitively, each ϕ_i represents an alternative execution of p. Because of these alternatives, no ϕ_i by itself is crucial to the success of p. In the terminology of nested transactions, each ϕ_i is a *non-vital* subtransaction of p.

As with nested transactions, the presence of alternative subtransactions allows transaction failure and rollback to be localized. This is possible because the effects of failure can be limited to a single subtransaction: if a subtransaction fails because of a logical error, then it can be undone and an alternative subtransaction can be executed.[1] In this way, we can undo the effects of a small part of a transaction without undoing the entire transaction (which is the normal procedure for classical transactions). This ability, known as *partial failure* or *partial rollback*, is particularly important for long-running transactions, since the likelihood of failure is high, and we do not want to undo a large quantity of work.

Example 2.3 (Save Points and Partial Rollback) Consider the following three rules:

$$parent \leftarrow task_1 \otimes choose \qquad\qquad choose \leftarrow task_2 \qquad\qquad choose \leftarrow task_3$$

These rules define a *parent* transaction having three subtransactions, $task_1$, $task_2$ and $task_3$, and a non-deterministic choice. The *parent* transaction commits if both $task_1$ and *choose* commit, and *choose* commits if $task_2$ or $task_3$ commit. Because *choose* has more than one possible execution, the point between $task_1$ and *choose* acts as both a choice point and a

[1] Even without alternatives, if a subtransaction fails because of a system error (*e.g.*, deadlock), then it can be undone and restarted by the transaction manager.

save point. That is, if an execution of *choose* aborts, then the state of the system can be rolled back to the choice point, from which a different execution of *choose* can be attempted.

As an example, consider a specific execution of the *parent* transaction. When *parent* is invoked, $task_1$ is immediately executed. If $task_1$ commits, then *choose* is invoked, which causes either $task_2$ or $task_3$ to be chosen non-deterministically. Suppose $task_2$ is chosen. If $task_2$ eventually aborts, then its effects must be undone; so, the database state and the program state are rolled back to the choice point. After rollback, $task_3$ is executed. If $task_3$ eventually commits, then *choose* commits, and the *parent* transaction commits. In this case, therefore, a local abort (of $task_2$) does not cause a global abort (of *parent*). Moreover, the choice point acts as a save point, so the effects of the abort are localized (to within the *choose* transaction). □

The next example shows how to achieve even finer control over subtransaction execution. It is also illustrates two forms of integration in Transaction Datalog: the integration of query specification with transaction control; and the integration of declarative constructs (such as queries) with procedural constructs (such as sequential composition). Like other transactions, queries are defined by sets of $\mathcal{TD}$ rules, as illustrated in Example 2.6.

Example 2.4 (Case Statements) Consider the following four rules:

$$trans \leftarrow task_1 \otimes query(X) \otimes case(X) \otimes task_5$$

$$case(a) \leftarrow task_2 \qquad\qquad case(b) \leftarrow task_3 \qquad\qquad case(c) \leftarrow task_4$$

These rules define a transaction, *trans*, involving five tasks, one query, and a simple case statement. When *trans* is invoked, $task_1$ is run. If $task_1$ succeeds, then $query(X)$ is evaluated. In this example, we suppose that the query returns a single answer, X. If $X = a$, then $case(a)$ is invoked, and $task_2$ is run. If $task_2$ succeeds, then $task_5$ is run. Likewise, if $X = b$ then $task_3$ is run, and if $X = c$ then $task_4$ is run. If the answer to the query is not a, b or c, then $case(X)$ fails, so *trans* fails, and the effects of $task_1$ are undone. □

The reader may have noticed that Examples 2.3 and 2.4 seem very much like logic programming. This should not be surprising since Transaction Datalog (like classical Datalog) has a logic-programming foundation. However, in classical logic programming, there are no updates, no transactions (classical or otherwise), and no atomicity, isolation, or rollback. A major contribution of Transaction Logic (and Transaction Datalog) is incorporating these features into the logic-programming (and deductive database) paradigm, and providing a logical semantics for them. These features are even more apparent in the next two subsections, where concurrency and isolation are introduced. A more-involved example of sequential transactions in $\mathcal{TD}$ is given in Section 4.1.

2.2 Concurrent Transactions

To combine transaction programs concurrently, $\mathcal{TD}$ includes a logical connective called *concurrent conjunction*, denoted $|$. Intuitively, if formulas ϕ_1 and ϕ_2 represent transaction programs, then the formula $\phi_1 \mid \phi_2$ represents their concurrent composition, that is, a program in which ϕ_1 and ϕ_2 execute concurrently in an interleaved fashion. As in most concurrent programming languages, programs in Transaction Datalog may communicate and synchronize themselves. This is possible because one program can read what another program writes. The database thus acts as the medium of communication.[2] Of course, when programs are executed in isolation (Section 2.3), communication can take place freely *within* individual programs, but not *between* them.

A transaction defined by concurrent conjunction succeeds if and only if each of its subtransactions succeed. More formally, the transaction $\phi_1 \mid \phi_2$ succeeds if and only if both ϕ_1 and ϕ_2 succeed (which is why $\mid$ is called concurrent *conjunction*). Like serial conjunction, concurrent conjunction leads immediately to relative commit. For instance, in the transaction $\phi_1 \mid \phi_2$, if subtransaction ϕ_1 fails, then the entire transaction fails, and subtransaction ϕ_2 is undone, even though it may have already succeeded (and committed). Thus, when a subtransaction commits, it only commits relative to its parent transaction. As a more concrete example, consider the following program:

$$transfer(fee, client, broker) \mid transfer(cost, client, seller) \qquad (2)$$

This is a concurrent version of Example 2.2, involving two money transfers. As in the sequential version, if either transfer fails, then both transfers are undone, and the database is restored to its initial state. Unlike the sequential version, either transfer can now start first, and neither is delayed by an artificially-imposed execution order. In particular, each transfer can execute as soon as the data items it needs are available (*i.e.*, not locked by other transactions). As another example, consider the composition of five transactions described in the third paragraph of Section 1.1. This composition is easily specified in Transaction Datalog by the following formula: $t_1 \otimes (t_2 \mid t_3 \mid t_4) \otimes t_5$. In this case, if one of transactions t_2, t_3 or t_4 fails, then the other two are aborted, and the effects of t_1 are also undone. Transaction t_5 is unaffected, since it had not been started when the failure occurred.

Concurrent programs in $\mathcal{TD}$ can cooperate by using the database to communicate and synchronize themselves. This idea is illustrated in Example 2.5 below. To convey the right intuition, we refer to formulas of the form $q_1 \otimes q_2 \otimes \cdots \otimes q_n$ as *sequential processes*, or simply as *processes*. The example also illustrates how concurrency in $\mathcal{TD}$ can be interpreted both procedurally and declaratively. The declarative semantics involves checking all possible interleavings of several processes, as described in Section 3. In contrast, the procedural semantics involves one process waiting for another process to perform an update, as described in Example 2.5. A more-involved example of cooperation between processes in $\mathcal{TD}$ is given in Section 4.2, where concurrent transactions are combined into a workflow.

Example 2.5 (Communication and Synchronization) The rules below define a process and two subprocesses. The subprocesses communicate with each other and synchronize the execution of several tasks.

$$\begin{array}{rcl} process & \leftarrow & processA \mid processB \\ processA & \leftarrow & taskA_1 \otimes ins.startB_2 \otimes taskA_2 \otimes startA_3 \otimes taskA_3 \\ processB & \leftarrow & taskB_1 \otimes startB_2 \otimes taskB_2 \otimes ins.startA_3 \otimes taskB_3 \end{array}$$

[2] Here, we are using the term "database" is a general sense that includes any kind of shared memory, as long as the information in it can be viewed as a set of tuples. In particular, the database can contain structures and access methods designed for efficient communication. For instance, some relations in the database could be a view of a set of message queues or communication channels, as described in [16].

The first rule defines the top-level process, which immediately splits into two subprocesses, called $processA$ and $processB$. The two subprocesses execute concurrently, but not independently. In particular, each subprocess executes three tasks, where $taskB_2$ cannot start until $taskA_1$ is finished, and $taskA_3$ cannot start until $taskB_2$ is finished. To see this, observe that while executing $taskA_1$ and $taskB_1$, the two subprocesses run concurrently without interacting with each other. However, when $processB$ completes $taskB_1$, it cannot start $taskB_2$ until the atom $startB_2$ is in the database, which only happens after $processA$ has executed $taskA_1$. In this way, the two subprocesses communicate, and $processB$ is synchronized with $processA$. Likewise, on completing $taskA_2$, $processA$ cannot start $taskA_3$ until the atom $startA_3$ is in the database, which only happens after $processB$ has executed $taskB_2$. In this way, the two subprocesses again communicate (in the reverse direction), and $processA$ is synchronized with $processB$. Observe that if $process$ is executed in isolation, then it is a transaction. However, because the two subprocesses communicate in both directions, they cannot be isolated from each other, so they are not subtransactions. □

Queries are transactions that do not update the database (*i.e.*, read-only transactions). Thus, in the absence of updates, transaction composition reduces to query composition, *i.e.*, the composition of simple queries into complex queries. In this case, serial and concurrent conjunction both reduce to classical conjunction, and Transaction Datalog reduces to classical Datalog. Formally, in the absence of updates, $\alpha \otimes \beta \;\equiv\; \alpha \mid \beta \;\equiv\; \alpha \wedge \beta$. This reduction leads to a seamless integration of procedural and declarative programming in $\mathcal{TD}$. Programs involving only queries are purely declarative. But, as updates are gradually introduced, programs gradually become procedural. In particular, conjunctive queries become sequential or concurrent programs, and union queries become non-deterministic programs.

Example 2.6 (Declarative Queries) The following rules of classical Datalog express the transitive closure of a binary relation, r:

$$tr(X,Y) \leftarrow r(X,Y) \qquad\qquad tr(X,Y) \leftarrow r(X,Z) \wedge tr(Z,Y)$$

These rules translate directly into Transaction Datalog in two ways.

Translation 1: $\quad tr(X,Y) \leftarrow r(X,Y) \qquad\qquad tr(X,Y) \leftarrow r(X,Z) \otimes tr(Z,Y)$

Translation 2: $\quad tr(X,Y) \leftarrow r(X,Y) \qquad\qquad tr(X,Y) \leftarrow r(X,Z) \mid tr(Z,Y)$

□

2.3 Isolation and Nested Transactions

As described above, concurrent programs in Transaction Datalog can interact and communicate with each other. Because communication can be two-way, executions of such programs need not be serializable [9], so $\mathcal{TD}$ programs need not be isolated transactions. To specify isolation, $\mathcal{TD}$ includes a logical modality called the *modality of isolation*, denoted $\odot$. Intuitively, the formula $\odot\phi$ means that program ϕ executes in isolation from all other concurrent programs. For instance, in the program $\phi_1 \mid (\odot\phi_2) \mid \phi_3$, the subprograms ϕ_1 and ϕ_3 may communicate with each other, but not with ϕ_2, which is an isolated transaction. As a special case, in the program $(\odot\phi_1) \mid (\odot\phi_2)$, the subprograms ϕ_1 and ϕ_2 execute in isolation from each other, and do not communicate. They must therefore execute as serializable transactions. In $\mathcal{TD}$, isolated transactions may be nested within other isolated transactions to arbitrary depth. For example, the program $\phi_1 \mid \odot(\phi_2 \mid \odot\phi_3)$ contains an isolated transaction, which in turn contains an isolated subtransaction. The transaction $\phi_2 \mid \odot\phi_3$ executes concurrently with, but in isolation from ϕ_1. Likewise, within this transaction, the subtransaction ϕ_3 executes concurrently with, but in isolation from ϕ_2.

As described earlier, logical rules are used to define named procedures and subroutines. In general, the body of a rule may use the three connectives $\otimes$, $\mid$ and $\odot$ in any combination. For instance, the formula $p \leftarrow (q_1 \otimes q_2) \mid \odot(r_1 \otimes r_2)$ is a legal rule. Intuitively, this rule says, "To execute procedure p, concurrently execute the programs $q_1 \otimes q_2$ and $r_1 \otimes r_2$, where the latter program must execute in isolation."

Example 2.7 (Financial Transactions: II) Consider the banking programs of Example 2.1, which transfer money between accounts. In the presence of concurrency, these programs must be modified to ensure that they execute as transactions. For instance, as is, there is nothing to prevent non-serializable behavior during two concurrent money transfers, as in program (2). We can use the modality of isolation to ensure serializability. We can also use concurrent conjunction to exploit intra-transaction concurrency, and increase the throughput of the transaction system. Here are the modified rules:

$$sell(Brkr, Client, Seller, Cost, Fee) \leftarrow \\ \odot[transfer(Fee, Client, Brkr) \mid transfer(Cost, Client, Seller)]$$

$$transfer(Amt, Acct_1, Acct_2) \leftarrow \odot[withdraw(Amt, Acct_1) \mid deposit(Amt, Acct_2)]$$

$$withdraw(Amt, Acct) \leftarrow \\ \odot[balance(Acct, Bal) \otimes Bal > Amt \otimes change\_balance(Acct, Bal, Bal - Amt)]$$

$$deposit(Amt, Acct) \leftarrow \odot[balance(Acct, Bal) \otimes change\_balance(Acct, Bal, Bal + Amt)]$$

$$change\_balance(Acct, Bal_1, Bal_2) \leftarrow del.balance(Acct, Bal_1) \mid ins.balance(Acct, Bal_2)$$

These rules define four isolated transactions—*sell*, *transfer*, *withdraw* and *deposit*—and one subroutine—*change_balance*. Observe that *withdraw* and *deposit* are nested within *transfer*, and two instances of *transfer* are nested within *sell*. In these rules, we have used concurrent composition where possible, although in some cases, we have used sequential composition because of dataflow within a rule. For instance, in the rule for *withdraw*, the account balance is retrieved and tested before it is updated. Note that the rule for *sell* simply turns program (2) into a named transaction. □

The depth of nesting in Transaction Datalog is not always static, as in Example 2.7, but can depend on the database. Dynamic nesting arises from recursion through isolation. Such recursions add no complications to the logical semantics of Transaction Datalog.

Example 2.8 (Dynamic Nesting) Suppose that r is a database relation with n tuples. Then, the rules below define a transaction $trans$ that spawns n concurrent instances of $task(x)$, one instance for each tuple x in relation r. Moreover, as they are spawned, successive tasks are nested more and more deeply within $trans$, so that the final task is nested $n-1$ levels deep.

$$trans \leftarrow r(X) \otimes del.r(X) \otimes [task(X) \mid \odot\, trans]$$
$$trans \leftarrow empty.r$$

The first rule is recursive. At each level of recursion, it non-deterministically chooses a tuple X from relation r, deletes it from the database, and then applies the task to the tuple by spawning $task(X)$ as a concurrent process. In addition, the rule calls itself recursively and in isolation; so, each recursive call to $trans$ is nested one level deeper than the previous call. The second rule halts the recursion after all the tuples have been deleted from relation r, *i.e.*, after $n-1$ recursive calls □

3 Syntax and Semantics

Recall that Transaction Datalog is a fragment of Transaction Logic, which is a general logic of state change [14, 16, 13, 19, 15]. Transaction Datalog therefore inherits the semantics of Transaction Logic, including its model theory and proof procedures, which have been published elsewhere [14, 16, 13, 19]. For convenience, this section develops a simplified version of that semantics, specialized for Transaction Datalog. The simplification comes from restricting Transaction Logic to relational databases and Horn-like rules without function symbols (in much the same way that classical Datalog is a restriction of classical logic). The simplified semantics is based on a logical inference system that describes the legal execution traces of a $\mathcal{TD}$ program.

It should also be mentioned that Transaction Logic (and thus Transaction Datalog) has an operational semantics based on a proof procedure with unification [16, 15, 13]. This procedure executes transactions, updates the database, and generates query answers, all as a result of proving theorems. Transactional features such as abort, rollback, and save-points are also handled by the proof procedure. This procedure is the foundation of our implementation [44].

3.1 Syntax

The language of Transaction Datalog includes three infinite enumerable sets of symbols: constant symbols $(a, b, c, ...)$, variables $(X, Y, Z, ...)$, and predicate symbols $(p, q, r, ...)$. We adopt the Prolog convention that variables begin in upper case, and constant symbols begin in lower case. As in classical Datalog, there are two sorts of predicate symbol: base and derived. In addition, for each base predicate, p, there are three special predicate symbols, denoted *empty.p*, *ins.p* and *del.p*. The first has arity zero, and the other two have the same arity as p. We define a *database state* to be a finite set of ground atomic formulas with base predicate symbols. We sometimes refer to a database state simply as a *database* or a *state*.

Definition 3.1 (Goals and Rules) A *goal* is a formula of the following form:

- an atomic formula;
- $(\phi_1 \otimes \phi_2 \otimes \cdots \otimes \phi_k)$ where $k \geq 0$ and each ϕ_i is a goal; or
- $(\phi_1 \mid \phi_2 \mid \cdots \mid \phi_k)$ where $k \geq 0$ and each ϕ_i is a goal; or
- $\odot\phi$ where ϕ is a goal.

A *rule* is a formula of the form $p \leftarrow \phi$, where ϕ is a goal, and p is an atomic formula with a derived predicate symbol. □

We use the symbol ( ) to denote the empty goal, *i.e.*, the goal consisting of 0 atoms. Intuitively, ( ) is a no-op, *i.e.*, a transaction that does nothing and leaves the database unchanged. When it used in a rule body, we write simply $p \leftarrow$ instead of $p \leftarrow ()$. In writing goals and rules, parentheses will be used only if needed to resolve ambiguity.

A *transaction base* is a set of rules. A *program* is a transaction base together with a goal. Intuitively, the goal defines the main procedure, and each rule in the transaction base defines a subroutine. When the transaction base is implicit, we sometimes refer to the goal as a program. A *transaction program* is a program whose main procedure executes in isolation, *i.e.*, has the form $\odot\phi$. In the literature [9, 40], a *transaction* is a particular execution of a transaction program. This paper uses the same definition, but when there is no confusion, we sometimes use "transaction" as an abbreviation for "transaction program."

3.2 Execution Traces

Concurrency in Transaction Datalog has an interleaving semantics. Intuitively, a $\mathcal{TD}$ program consists of a number of concurrent processes, where each process generates 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 legal interleavings is determined partly by the need for subtransactions to execute in isolation, and partly by the need for other activities to execute cooperatively. As an example of the latter, suppose that one process writes data that another process must read; then the write operation must come *before* the read operation in the interleaved sequence. These needs are specified by $\mathcal{TD}$ programs.

In an interleaving semantics, only one program executes at a time, while all concurrent programs are suspended. To model this behavior, $\mathcal{TD}$ records the state of the database whenever a program is suspended or awakened. Formally, an execution of a program, ϕ, is represented as a finite sequence of pairs, $\mathbf{D}_1\mathbf{D}_2, \mathbf{D}_3\mathbf{D}_4, \mathbf{D}_5\mathbf{D}_6, ..., \mathbf{D}_{n-1}\mathbf{D}_n$, which we call an *execution trace*, or simply an *execution* or a *trace*. In this sequence, each pair $\mathbf{D}_i\mathbf{D}_{i+1}$ represents a period of uninterrupted execution of program ϕ during which ϕ changes the database from state $\mathbf{D}_i$ to $\mathbf{D}_{i+1}$. Between adjacent pairs, ϕ is suspended and other programs execute. Thus, initially ϕ changes the database from state $\mathbf{D}_1$ to $\mathbf{D}_2$. Then, ϕ is suspended, while other programs change the database from state $\mathbf{D}_2$ to $\mathbf{D}_3$. Then, ϕ is awakened and changes the database from $\mathbf{D}_3$ to $\mathbf{D}_4$. Then, ϕ is suspended again, while other programs change the database from $\mathbf{D}_4$ to $\mathbf{D}_5$. This process of execution and suspension continues until ϕ terminates, leaving the database in state $\mathbf{D}_n$. For example, the sequence $\{a\}\,\{ab\},\ \{d\}\,\{cd\}$ is an execution trace of the program $ins.b \otimes ins.c$. That is, starting from state $\{a\}$, the program first inserts b, changing the database to state $\{ab\}$. Then, the program is suspended, and other programs change the database to state $\{d\}$. Finally, the original program is re-awakened, and it inserts the atom c, leaving the database in state $\{cd\}$.

If a program is isolated, then its execution is not interleaved with that of any other programs. It should therefore execute continuously, without interruption or suspension. An execution trace of an isolated program thus consists of a single database pair, $\mathbf{D}_1\mathbf{D}_2$. For example, the pair $\{a\}\,\{abc\}$ is an execution trace of the program $\odot(ins.b \otimes ins.c)$. That is, starting from state $\{a\}$, the program inserts the atoms b and c, leaving the database in state $\{abc\}$. Transactions always execute in isolation, so in $\mathcal{TD}$, each execution of a transaction is represented by a single database pair. One consequence of this idea is that a concurrent execution of several transactions is equivalent to a serial execution. For instance, if ϕ_1 and ϕ_2 are $\mathcal{TD}$ programs, then a correct execution of $(\odot\phi_1) \mid (\odot\phi_2)$ is equivalent to an execution of $\phi_1 \otimes \phi_2$ or $\phi_2 \otimes \phi_1$.

We are *not* saying here that to achieve isolation, transactions must be executed serially. Rather, a program that executes in isolation must behave *as if* it were not interleaved with any other programs. As a special case, a concurrent execution of transactions must have the same effect as a serial execution; *i.e.*, transactions must be serializable, which is the normal understanding in database concurrency control [9]. Our semantics therefore specifies the *effects* of a $\mathcal{TD}$ program, but not its actual execution inside a DBMS. In fact, inside a DBMS, concurrent programs may be executed in parallel, rather than in an interleaved fashion. For instance, suppose that predicates p and q are stored on different disks. Then, when the transaction $ins.p(a) \mid ins.q(b)$ is executed, the elementary updates $ins.p(a)$ and $ins.q(b)$ can be executed *simultaneously*. On the other hand, if p and q are stored on the same disk, then $ins.p(a)$ and $ins.q(b)$ must be executed serially, in some order. In either case, the effect is the same: to insert the atoms $p(a)$ and $q(b)$ into the database. The details of how and when concurrent operations are actually executed is an implementation issue, and is beyond the scope of this paper.

With the above model of execution, we can develop a simple semantics for the three logical connectives $\otimes$, $\mid$ and $\odot$. The semantics is defined in terms of three operations on execution traces: *concatenation*, *interleaving* and *reduction*. The first two are familiar list operations. For example, the concatenation of lists $[a, b, c]$ and $[x, y, z]$ is the list $[a, b, c, x, y, z]$. An interleaving of two lists, L_1 and L_2, is any list composed of the elements of L_1 and L_2 that preserves the relative order of the elements in each list. For example, the two lists $[a, b]$ and $[x, y]$ have six interleavings:

$$[a,b,x,y] \qquad [a,x,b,y] \qquad [a,x,y,b] \qquad [x,a,b,y] \qquad [x,a,y,b] \qquad [x,y,a,b]$$

We use concatenation and interleaving to model serial and concurrent conjunction, respectively. Intuitively, suppose that $\overline{\mathbf{D}}_1$ is an execution of ϕ_1, and $\overline{\mathbf{D}}_2$ is an execution of ϕ_2. Then, the concatenation of $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$ is an execution of $\phi_1 \otimes \phi_2$, and any interleaving of $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$ is an execution of $\phi_1 \mid \phi_2$.

Unlike concatenation and interleaving, which are general list operations, reduction is specific to execution traces.

Definition 3.2 (Reduction) The execution trace $[\mathbf{D}_1\mathbf{D}'_1, \mathbf{D}_2\mathbf{D}'_2, ...\mathbf{D}_n\mathbf{D}'_n]$ is *reducible* if $\mathbf{D}'_i = \mathbf{D}_{i+1}$ for $1 \leq i \leq n$. In this case, $[\mathbf{D}_1\mathbf{D}'_n]$ is the *reduction* of the trace. □

Thus $[\mathbf{D}_1\mathbf{D}_2, \mathbf{D}_2\mathbf{D}_3, \mathbf{D}_3\mathbf{D}_4]$ is reducible, and its reduction is $[\mathbf{D}_1\mathbf{D}_4]$. Intuitively, if a program has a reducible execution trace, then the database does not change when the program is suspended. The suspensions are therefore unnecessary, and the program could execute continuously, without interruption. The reduced trace therefore represents another possible execution of the program. In fact, it represents an *isolated* execution, *i.e.*, an execution that is not interleaved with the executions of other programs. Intuitively, if $[\mathbf{D}_1\mathbf{D}_2, \mathbf{D}_2\mathbf{D}_3, \mathbf{D}_3\mathbf{D}_4]$ is an execution of ϕ, then $[\mathbf{D}_1\mathbf{D}_4]$ is an execution of $\odot\phi$.

3.3 Logical Inference

This section develops a declarative semantics for $\mathcal{TD}$. The development is based on a logical inference system that specifies the legal execution traces of a $\mathcal{TD}$ program. In [16], an equivalent, model-theoretic semantics is developed, along with a practical proof procedure based on unification.

The inference system below manipulates expressions of the form $\mathbf{P} : \overline{\mathbf{D}} \vdash \phi$, called *sequents*. Here, $\mathbf{P}$ is a transaction base (intuitively, a set of program definitions), ϕ is a ground goal (intuitively, a procedure invocation), and $\overline{\mathbf{D}}$ is an execution trace. This sequent means that $\overline{\mathbf{D}}$ is an execution trace of program ϕ. The inference system itself is a collection of axioms and inference rules. Each inference rule consists of several sequents, and has the following interpretation: if the sequent(s) above the horizontal line can be derived, then the sequent below the line can also be derived. Based on the axiom sequents, the system uses the inference rules to derive more-and-more sequents. In this system, all execution traces are *safe*. That is, programs do not expand the data domain during execution, and in particular, they do not create new domain values and insert them into the database (so-called value invention [1]). Safety is guaranteed by defining inference with respect to a *finite* data domain.[3]

Definition 3.3 (Inference System) Let *dom* be a finite set of constant symbols, called the *data domain*. Then $\Im(dom)$ is the system of axioms and inference rules below, where each sequent contains only those constants in *dom*. Here, $\mathbf{P}$ is a transaction base, $\mathbf{D}$ is a database, $\overline{\mathbf{D}}$ is an execution trace, q is a ground atomic formula, and ϕ is a ground goal.

Axioms:

1. *Elementary Queries:*

$$\mathbf{P} : \mathbf{DD} \vdash ()$$

$$\mathbf{P} : \mathbf{DD} \vdash q \qquad \textit{if} \quad q \in \mathbf{D}$$

$$\mathbf{P} : \mathbf{DD} \vdash \textit{empty.r} \qquad \textit{if} \quad \mathbf{D} \textit{ contains no atoms with predicate symbol } r$$

2. *Elementary Updates:*

$$\mathbf{P} : \mathbf{D}_1\mathbf{D}_2 \vdash \textit{ins.q} \qquad \textit{if} \quad \mathbf{D}_2 = \mathbf{D}_1 + \{q\}$$

$$\mathbf{P} : \mathbf{D}_1\mathbf{D}_2 \vdash \textit{del.q} \qquad \textit{if} \quad \mathbf{D}_2 = \mathbf{D}_1 - \{q\}$$

Inference Rules:

3. *Subroutines:*

$$\frac{\mathbf{P} : \overline{\mathbf{D}} \vdash \phi}{\mathbf{P} : \overline{\mathbf{D}} \vdash q} \qquad \textit{if} \quad q \leftarrow \phi \quad \textit{is a ground instantiation of a rule in } \mathbf{P}.$$

4. *Sequential Composition:*

$$\frac{\mathbf{P} : \overline{\mathbf{D}}_1 \vdash \phi_1 \qquad \mathbf{P} : \overline{\mathbf{D}}_2 \vdash \phi_2}{\mathbf{P} : \overline{\mathbf{D}}_3 \vdash \phi_1 \otimes \phi_2} \qquad \textit{if} \quad \overline{\mathbf{D}}_3 \textit{ is the concatenation of } \overline{\mathbf{D}}_1 \textit{ and } \overline{\mathbf{D}}_2.$$

[3] In contrast to Transaction Datalog, the semantics of Transaction Logic is defined with respect to an *infinite* data domain, including both constant and function symbols [14, 16, 13, 15]. In this more-general setting, value invention is possible, and safety is guaranteed by syntactic restrictions.

5. *Concurrent Composition:*

$$\frac{\mathbf{P} : \overline{\mathbf{D}}_1 \vdash \phi_1 \qquad \mathbf{P} : \overline{\mathbf{D}}_2 \vdash \phi_2}{\mathbf{P} : \overline{\mathbf{D}}_3 \vdash \phi_1 \mid \phi_2} \qquad \textit{if } \overline{\mathbf{D}}_3 \textit{ is an interleaving of } \overline{\mathbf{D}}_1 \textit{ and } \overline{\mathbf{D}}_2.$$

6. *Isolation:*

$$\frac{\mathbf{P} : \overline{\mathbf{D}}_1 \vdash \phi}{\mathbf{P} : \overline{\mathbf{D}}_2 \vdash \odot\, \phi} \qquad \textit{if } \overline{\mathbf{D}}_1 \textit{ reduces to } \overline{\mathbf{D}}_2.$$

□

Each axiom and inference rule in Definition 3.3 has a simple, intuitive interpretation. For instance, axioms of type 1 all have the form $\mathbf{P} : \mathbf{DD} \vdash \phi$. Here, the execution trace is a single database pair, $\mathbf{DD}$, in which the initial and final states are the same, $\mathbf{D}$, which means that ϕ is a read-only transaction (*i.e.*, a query). The first axiom defines the empty goal ( ), which is a transaction that does nothing and always succeeds. The second axiom defines simple queries that ask whether a given atom, q, is in the database. The third axiom defines queries that ask whether a given relation, r, is empty.

Axioms of type 2 all have the form $\mathbf{P} : \mathbf{D}_1\mathbf{D}_2 \vdash \phi$. Here, the execution trace is a single database pair, $\mathbf{D}_1\mathbf{D}_2$, in which the initial and final states of the database may be different. This means that ϕ is an updating transaction that changes the database from state $\mathbf{D}_1$ to $\mathbf{D}_2$. The first axiom says that transaction *ins.q* changes the database from state $\mathbf{D}$ to state $\mathbf{D} + \{q\}$. Likewise, the second axiom says that transaction *del.q* changes the database from state $\mathbf{D}$ to state $\mathbf{D} - \{q\}$. The following sequents are instances of these two axioms:

$$\mathbf{P} :\ \{p\}\,\{pq\} \vdash\ \mathit{ins.q} \qquad\qquad \mathbf{P} :\ \{pq\}\,\{q\} \vdash\ \mathit{del.p} \tag{3}$$

The four inference rules are also straightforward. For instance, suppose that $\overline{\mathbf{D}}_1$ is an execution of ϕ_1, and $\overline{\mathbf{D}}_2$ is an execution of ϕ_2. Then, rule 4 says that the concatenation of $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$ is an execution of $\phi_1 \otimes \phi_2$. Likewise, rule 5 says that any interleaving of $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$ is an execution of $\phi_1 \mid \phi_2$. Thus, the following sequent can be derived from sequents (3) using inference rule 4:

$$\mathbf{P} :\ \{p\}\,\{pq\},\ \{pq\}\,\{q\} \vdash\ \mathit{ins.q} \otimes \mathit{del.p} \tag{4}$$

Rule 6 says that if $\overline{\mathbf{D}}$ is an execution of ϕ, then the reduction of $\overline{\mathbf{D}}$ is an execution of $\odot\phi$, assuming that $\overline{\mathbf{D}}$ is reducible. Thus, the following sequent can be derived from sequent (4) using inference rule 6:

$$\mathbf{P} :\ \{p\}\,\{q\} \vdash\ \odot\,(\mathit{ins.q} \otimes \mathit{del.p}) \tag{5}$$

Inference rule 3 uses the rules in the transaction base, $\mathbf{P}$. Recall that each rule represents a procedure, where the rule head is the procedure name, and the rule body is the procedure definition. Variables in the rule represent parameters of the procedure, and are instantiated at run time. Intuitively, inference rule 3 says that if $\overline{\mathbf{D}}$ is an execution of an instantiated procedure body, ϕ, then it is also an execution of the instantiated procedure name, q. For instance, if $r \leftarrow \odot(\mathit{ins.q} \otimes \mathit{del.p})$ is a ground instantiation of a rule in $\mathbf{P}$, then the following sequent can be derived from sequent (5) using inference rule 3:

$$\mathbf{P} :\ \{p\}\,\{q\} \vdash\ r$$

Example 3.4 (Intra-Transaction Concurrency) Let $\mathbf{P}$ be the following transaction base:

$$r \leftarrow \odot(p \mid q) \qquad\qquad p \leftarrow ins.a \otimes ins.b \qquad\qquad q \leftarrow ins.c \otimes ins.d$$

Then, program r executes as an isolated transaction with internal concurrency. Specifically, by the first rule, r spawns two concurrent processes, p and q. The executions of p and q may be interleaved, but the execution of r itself must take place in isolation, and cannot be interleaved with the executions of other programs, as specified by the $\odot$ modality. By the second rule, p is a sequential process that inserts two atoms into the database, one after the other. Likewise for q. The executions of p and q may be interleaved in several possible ways, and the deduction below illustrates one possibility. It also shows that transaction r can transform the database from the empty state $\{\,\}$ to state $\{abcd\}$. In this deduction, each sequent is derived from the sequent immediately above it, unless otherwise specified.

Elementary updates:

1.	$\mathbf{P} : \{\,\}\,\{a\} \vdash ins.a$	*by Axiom 2,*
2.	$\mathbf{P} : \{a\}\,\{ac\} \vdash ins.c$	*by Axiom 2,*
3.	$\mathbf{P} : \{ac\}\,\{abc\} \vdash ins.b$	*by Axiom 2,*
4.	$\mathbf{P} : \{abc\}\,\{abcd\} \vdash ins.d$	*by Axiom 2,*

Inferring p:

5.	$\mathbf{P} : \{\,\}\,\{a\}, \{ac\}\,\{abc\} \vdash ins.a \otimes ins.b$	*by Rule 4, using sequents* 1 *and* 3,
6.	$\mathbf{P} : \{\,\}\,\{a\}, \{ac\}\,\{abc\} \vdash p$	*by Rule 3, using* $p \leftarrow ins.a \otimes ins.b$,

Inferring q:

7.	$\mathbf{P} : \{a\}\,\{ac\}, \{abc\}\,\{abcd\} \vdash ins.c \otimes ins.d$	*by Rule 4, using sequents* 2 *and* 4,
8.	$\mathbf{P} : \{a\}\,\{ac\}, \{abc\}\,\{abcd\} \vdash q$	*by Rule 3, using* $q \leftarrow ins.c \otimes ins.d$,

Inferring r:

9.	$\mathbf{P} : \{\,\}\,\{a\}, \{a\}\,\{ac\}, \{ac\}\,\{abc\}, \{abc\}\,\{abcd\} \vdash p \mid q$	*by Rule 5, using sequents* 6 *and* 8,
10.	$\mathbf{P} : \{\,\}\,\{abcd\} \vdash \odot(p \mid q)$	*by Rule 6,*
11.	$\mathbf{P} : \{\,\}\,\{abcd\} \vdash r$	*by Rule 3, using* $r \leftarrow \odot(p \mid q)$.

□

4 Specifying and Coordinating Transactions

This section illustrates two facets of Transaction Datalog: (i) specifying the effects of a complex transaction, and (ii) coordinating the execution of a set of transactions. The ability to do both these tasks, and to integrate them seamlessly, is one property that distinguishes Transaction Datalog from other transaction languages.

4.1 Specifying Complex Transactions

The example below defines a transaction program that searches for a Hamiltonian path in a graph. The example illustrates several properties of $\mathcal{TD}$. One property is its computational power. Since Hamiltonian Path is an NP-complete problem, the data complexity of $\mathcal{TD}$ is at least NP-hard. Another property is non-determinism. For each initial database state, the transaction can terminate at many different final states, depending on which Hamiltonian path is found. In fact, the number of possible final states is equal to the number of distinct Hamiltonian paths in the graph, which can be exponential. Finally, the example also illustrates transaction failure and rollback, and the localized abort and recovery properties of nested transactions.

The program can be viewed as a greedy algorithm followed by a test and possible transaction abort. Each execution of the program constructs a path through the graph in a greedy fashion. If the path is not Hamiltonian, then the transaction aborts, and is rolled back and restarted. After each restart, the transaction constructs a different path. The program succeeds (and commits) if and only if some execution constructs a Hamiltonian path. It should be noted that the aborts and rollbacks are partial, not total. This is possible because the transaction has been programmed as a sequence of subtransactions, and rollback can be limited to those subtransactions at the end of the sequence. Each subtransaction adds an edge to the end of the path. When the path cannot be extended any further, the youngest subtransactions are rolled back, and new ones are started. In this way, edges are automatically removed from the end of the path, and new ones are added.

Example 4.1 (Hamiltonian Path) Suppose the database stores a directed graph. That is, the atom $node(x)$ is in the database if x is a node in the graph, and the atom $edge(x, y)$ is in the database if the graph has an edge from node x to node y. We want to determine if the graph has a Hamiltonian path, and if so, we want to mark all the edges on one such path (chosen non-deterministically). To mark the edge from x to y, we add the atom $mark(x, y)$ to the database. The following rules do exactly this:

$$path \leftarrow node(N) \otimes del.node(N) \otimes extend(N) \otimes ins.node(N)$$
$$extend(N_1) \leftarrow choose(N_1, N_2) \otimes del.node(N_2) \otimes extend(N_2) \otimes ins.node(N_2)$$
$$extend(N) \leftarrow empty.node$$
$$choose(N_1, N_2) \leftarrow edge(N_1, N_2) \otimes node(N_2) \otimes ins.mark(N_1, N_2)$$

If the database stores a graph, then the transaction $path$ succeeds if and only if the graph has a Hamiltonian path. Moreover, when it succeeds, the transaction non-deterministically chooses some Hamiltonian path in the graph, and marks each of its nodes, as described below. □

The details of Example 4.1 are best explained in terms of top-down inference. During inference the rules construct a path in the graph one node at a time. The first rule chooses a node N to start the path. The second rule is then applied recursively. It chooses a node N_2 connected by an edge to N_1, the last node in the growing path. Note that after N_2 is chosen, it is deleted from the graph, so that it will not be chosen again. The third rule says that a Hamiltonian path has been found when no more nodes are left in the graph, *i.e.*, when every node has been chosen (and deleted) exactly once. The actual choosing of nodes is done by the last rule, which also marks the edge as a member of the growing path. Note that each node choice is non-deterministic, so the program effectively chooses a path in the graph non-deterministically. The program succeeds if and only if some such path is Hamiltonian.

Observe that if a Hamiltonian path is found, then as the recursion unwinds, the first two rules re-insert each node back into the graph. This restores the nodes, so that they are not destroyed by the computation. If no Hamiltonian path is found, then the *path* transaction fails, so the database is automatically restored to its initial state.

4.2 Coordinating the Execution of Many Transactions

The example below simulates a simple production workflow. Intuitively, a workflow is a network of concurrent activities whose execution must be coordinated [38]. Ensuring their proper coordination and timely execution is known as workflow management.

In so-called *production* workflows, activities are organized into factory-like production lines [45]. Many production workflows are organized around work items of some kind, which the workflow activities operate on. Examples of work items include insurance claims, loan applications, and laboratory samples. Often, workflows are represented as directed graphs, where each node represents a workflow activity, and each arc represents a flow of work items from one activity to another. The example below deals with linear graphs, but can be extended to arbitrary graphs without much difficulty. The workflow model in the example is based on workflows at the Whitehead Institute/MIT Center for Genome Research, in which workflows are used to carry out the tens of millions of biological experiments needed to map and sequence the human genome [10, 67].

Example 4.2 (Workflow) Consider a workflow consisting of a sequence of three activities. The workflow processes a stream of work items. Each item is processed first by Activity 1, then by Activity 2, and finally by Activity 3. Although the activities appear to be sequential, they execute concurrently, since at any given time, each activity is processing a different item.

The three activities are not synchronized, and they may process items at different rates. To allow for this, the production line has four baskets for holding work items. The baskets and activities are arranged sequentially as follows:

basket 1 — Activity 1 — basket 2 — Activity 2 — basket 3 — Activity 3 — basket 4

Each activity takes an item out of the basket to its left, processes it, and then puts it in the basket to its right. Each item starts in basket 1, passes through baskets 2 and 3, and ends up in basket 4. A basket may hold any number of work items, and we assume that initially basket 1 has many (hundreds, thousands or millions) of work items in it, while the other baskets are initially empty. To complicate the situation, we suppose that Activity 2 is so slow that we need two instances of it working concurrently to maintain throughput. We refer to these two instances as Activity 2a and Activity 2b. They both access baskets 2 and 3.

To represent workflows, we use predicate symbols of the form $basket_j$, $task_j$, $activity_j$ and $instance_j$. In addition, each work item has an identifier, which we represent as a constant symbol. Intuitively, the atom $basket_j(w)$ means that work item w is in basket j. The atom $task_j(w)$ is a transaction that simulates the effect of Activity j processing item w. We assume that definitions of these transactions are supplied to us. The atom $activity_j$ is a process that executes $task_j$ over-and-over again on different work items, until all the items in the database are processed. The atom $instance_j(i)$ represents an instance of Activity j, where $i \in \{a, b\}$ and identifies the instance. In addition, we use several helper predicates:

get_j, put_j and $done_j$. Intuitively, $get_j(w)$ is a transaction that gets work item w from basket j. Likewise, $put_j(w)$ puts work item w into basket j. The predicate $done_j$ is used to signal that activity j has terminated. To terminate an activity, we look for an empty basket.

The following rules define the transaction *simulate*, which simulates a complete execution of the workflow.

$$
\begin{array}{rcl}
simulate & \leftarrow & activity_1 \mid activity_2 \mid activity_3 \\
activity_2 & \leftarrow & instance_2(a) \mid instance_2(b) \\
activity_1 & \leftarrow & get_1(W) \otimes [\odot task_1(W)] \otimes put_2(W) \otimes activity_1 \\
activity_1 & \leftarrow & empty.basket_1 \otimes ins.done_1 \\
instance_2(I) & \leftarrow & get_2(W) \otimes [\odot task_2(W)] \otimes put_3(W) \otimes instance_2(I) \\
instance_2(I) & \leftarrow & done_1 \otimes empty.basket_2 \otimes ins.done_2(I) \\
activity_3 & \leftarrow & get_3(W) \otimes [\odot task_3(W)] \otimes put_4(W) \otimes activity_3 \\
activity_3 & \leftarrow & done_2(a) \otimes done_2(b) \otimes empty.basket_3 \otimes ins.done_3 \\
get_j(W) & \leftarrow & \odot[basket_j(W) \otimes del.basket_j(W)] \\
put_j(W) & \leftarrow & ins.basket_j(W)
\end{array}
$$

The first rule spawns three concurrent processes, one for each activity. The second rule then splits one of these processes into two concurrent subprocesses, one for each instance of Activity 2. There are now four concurrent processes, $activity_1$, $instance_2(a)$, $instance_2(b)$ and $activity_3$. Each process is defined by two rules. The first rule for Activity j chooses a work item W in basket j, executes task j on the item, and then moves the item to basket $j+1$. The rule then calls itself recursively, to process another work item. The second rule for Activity j terminates the recursion when there are no more work items left to process. The first activity terminates when its basket is empty. Each other activity terminates when its basket is empty and the previous activity has terminated. An activity with more than one instance terminates when all its instances have terminated. Activity j signals that it has terminated by inserting the atom $done_j$ in the database.

Since each task is a transaction, each occurrence of the atom $task_j(W)$ is enclosed in a modality of isolation. The body of the rule defining get_j is also executed in isolation, so that different instances of Activity j do not accidentally get the same work item from basket j. □

5 Related Work

This section compares and contrasts Transaction Datalog with other languages in the literature. We have divided the comparison into several broad areas. Due to space limitations, we have limited most of the comparisons to formalisms involving concurrency. In addition, T$\mathcal{D}$ can be compared to the numerous logics for representing action. These include dynamic logic, process logic, action logic, algorithmic logic, procedural logic, the event calculus, the situation calculus, and many others. However, none of these formalisms provide concurrency and communication, none provide composition of transaction programs, and none can model nested transactions. In addition, many have no notion of database state or declarative query, many are propositional, and many are simply inappropriate for database applications. An extensive comparison of these formalisms with the sequential version of Transaction Logic can be found in [15, 14].

Transaction Languages: Broadly speaking, the theoretical literature has explored two kinds of transaction language, in order to address two different problems. In the first approach, the user specifies the effects of individual transactions; and in the second approach, he coordinates the execution of a set of transactions. We shall refer to these two approaches as *specification* and *coordination*, respectively. In software-engineering terms, these two approaches correspond to "programming in the small" and "programming in the large," respectively [31, 29].

The specification approach implicitly focuses on classical transactions. The problem is to develop a high-level language for specifying database queries and updates, and to establish its theoretical properties, such as formal semantics, data complexity, and expressive power. Numerous languages with logical, algebraic and procedural semantics have been developed. Like SQL and relational algebra, these languages are often related to first-order predicate logic. Typical results are, "Language L1 expresses more transactions than language L2," and "The data complexity of language L1 is complete for PSPACE." Relationships between transactions are not an issue here; so concurrency, communication, isolation, abort and rollback are not addressed. Formally, these issues are abstracted away, and only the effects of transactions are considered. These languages therefore model a transaction as a mapping from databases to databases. Developments in this area include the procedural and declarative transaction languages of Abiteboul and Vianu [3, 2, 1], the procedural language QL of Chandra and Harel [23, 24],[4] Dynamic Prolog [50, 73], LDL [57, 58], and numerous other languages (*e.g.*, [20, 25, 62, 22, 30]). Detailed discussion of these works can be found in [17, 14, 15].

The coordination approach focuses on non-classical transactions. The problem is to develop a high-level language for combining a set of tasks into a larger application or software system. The focus is on relationships between tasks. Typical problems are to specify intertask dependencies, including data-flow and control flow, and to schedule and coordinate the execution of tasks. A typical control dependency is, "Task T2 cannot start until task T1 has committed;" and a typical data dependency is, "Task T2 can start if task T1 returns a value greater than 25" [63]. Specifying database updates and queries is not an issue here. Formally, the effects of tasks are abstracted away, and only the relationships between tasks are considered. Typically, these languages model a classical transaction as a finite automaton with a small number of states such as "start," "commit" and "abort." Temporal constraints between the states of different automata are then specified in a propositional logic. Developments in this area include ACTA [26, 27], proposals for Third Generation TP Monitors [29], approaches based on temporal logic [6] and event algebras [66, 65], and numerous other works (*e.g.*, [46, 36]).

In this paper, we addressed both issues, and integrated them into a single language. Specifically, Transaction Datalog can specify the effects of classical and non-classical trans-

[4] Although presented as a query language, QL is even more natural as an update language.

actions, and it can compose simple transaction programs into complex ones. For instance, $\mathcal{TD}$ can specify queries (Example 2.6), updates (Examples 2.1 and 4.1), and nested transactions (Example 2.7). Given a set of transaction programs, $\mathcal{TD}$ can impose a control structure on them (Examples 2.2 and 2.4), co-ordinate their execution (Examples 2.5 and 4.2), and nest them to arbitrary depth (Example 2.8). The programs themselves can execute sequentially, concurrently and non-deterministically, they can execute in isolation, and they can cooperate with each other by communicating and synchronizing.

We do not claim that Transaction Datalog deals with *all* the issues of transaction co-ordination. For instance, to simulate some non-classical transaction models, we must augment $\mathcal{TD}$ with negation-as-failure, as mentioned earlier. In addition, a complete treatment must deal with several other issues, including non-relational databases and legacy systems. Many of these issues are dealt with in the more general framework of *Transaction Logic* [14, 16, 13, 19, 15]. Transaction Datalog is a fragment of Transaction Logic. We have focussed on this fragment because it captures many essential properties of the general logic in a simpler setting. In addition, it has interesting complexity properties [12], and it greatly extends the capabilities of deductive database systems.

Although not the subject of this paper, several results on the complexity and expressibility of $\mathcal{TD}$ are worth mentioning [12]. For instance, the data complexity of $\mathcal{TD}$ is complete for RE. In addition, $\mathcal{TD}$ is expressively complete; *i.e.*, it expresses all computable generic transactions. Unlike other update languages, $\mathcal{TD}$ achieves expressive completeness with a *fixed* database schema and a *fixed* data domain, *i.e.*, *without* inventing new domain values during inference. Instead, expressive completeness comes entirely from the interaction of concurrency and updates. Indeed, removing concurrency from $\mathcal{TD}$ causes a dramatic loss of completeness, reducing the data complexity from RE to EXPTIME.

Concurrent Logic Programming: There has been considerable research on concurrency in the logic programming community. However, this work has focussed on the implementation of concurrency and on communication via shared variables. In particular, there has been no emphasis on logical semantics, database updates, or database transactions. Transaction Datalog and Transaction Logic therefore make a two-fold contribution to logic programming. First, they extend the logic programming paradigm with a host of transactional notions, including atomicity, isolation, rollback, and subtransaction hierarchies. Second, they integrate concurrency, communication and updates into a purely logical framework, including a natural model theory and a sound-and-complete proof theory [16].

This integration presents interesting possibilities for concurrent logic programming (CLP). For instance, concurrent processes can now communicate via the database, since one process can read what another process writes. This form of communication leads to a programming style that is very different from that of existing CLP languages [64]. In such languages, concurrent processes communicate via shared variables and unification. This kind of communication is orthogonal to communication via the database. Both are possible in $\mathcal{TD}$. Implementations of $\mathcal{TD}$ may therefore adopt many of the techniques of shared-variable communication developed for CLP. However, this possibility is *not* the focus of our work. Instead, we focus on concurrent processes that interact and communicate via the database. Indeed, one of the novelties of $\mathcal{TD}$ is that it provides a logical foundation for exactly this kind of interaction.

Process Algebras: These are a family of algebraic systems for modeling concurrent communicating processes. They include Milner's *Calculus of Communicating Systems* (CCS) [51], and Hoare's *Communicating Sequential Processes* (CSP) [41], among others. Transaction Datalog and process algebras use very different formal frameworks. This difference is most easily seen in terms of COSY [47], an early algebraic approach to modeling concurrent processes. COSY is an extension of regular expressions, while Transaction Datalog is an extension of deductive databases. Process algebras have since developed into equational theories, but the formal differences with $\mathcal{TD}$ remain the same.

The main conceptual difference between process algebras and Transaction Datalog is that process algebras are high-level models of shared-nothing systems, while Transaction Datalog is a high-level model of shared-memory systems, especially database systems with transaction processing. For instance, process algebras explicitly reject the notion of processes interacting via shared memory (such as a database) [52]. Instead, each process has its own local memory, and it interacts with other processes via synchronized communication. In contrast, Transaction Datalog is explicitly intended for database transactions, *i.e.*, processes that interact with a shared database. As such, it provides high-level primitives for database functions such as declarative queries, subtransaction hierarchies, serializable execution, transaction abort and rollback, etc. This difference in intent is reflected by differences in semantics: process algebras emphasis synchronized communication, while Transaction Datalog emphasizes database states.

Transaction Datalog integrates processes and data. It therefore unifies two previously disparate views of information systems and workflow management: the process-oriented view, and the data-oriented view. The former view is embodied in business processes and process algebras, while the latter view is embodied in database systems and query languages. As the examples in this paper illustrate, programs in Transaction Datalog can take either point of view, or a combination of both. For instance, Example 2.6 is data-oriented, while Example 2.3 is process-oriented. Other examples in the paper are at different points between these two extremes. Examples 2.1, 2.7 and 4.1 are towards the data-oriented end of the spectrum, while Examples 2.4, 2.5 and 4.2 are towards the process-oriented end, and Example 2.2 is in the middle.

A comparison of Transaction Logic and the recently developed pi calculus [53, 54] can be found in [16].

Concurrent Dynamic Logic: In [61, 60], Peleg develops Concurrent Dynamic Logic (CDL). There are two versions of CDL. The version developed in [61] is very different from $\mathcal{TD}$. 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 [11], in which hypothetical databases represent the states of the various (non-communicating) concurrent processes in an alternating computation. The version of CDL developed in [60] 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 $\mathcal{TD}$. CDL was intended to *reason about* what is true during program execution, while $\mathcal{TD}$ was designed to *execute* logically programmed 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 $\mathcal{TD}$, 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 $\mathcal{TD}$ and CDL is in the nature of states. In $\mathcal{TD}$, states are relational databases that can be queried and updated. In contrast, CDL focuses on the internal states of executing programs, while the notion of a database state is entirely missing.

Acknowledgements:

Transaction Logic was developed in collaboration with Michael Kifer [14, 16, 13, 19, 15]. Thanks go to David Toman and Michael Kifer for their comments on this paper. This work was supported in part by a Research Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] S. Abiteboul. Updates, a new frontier. In *Intl. Conference on Database Theory*, pages 1–18, 1988.

[2] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41:181–229, 1990.

[3] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.

[4] D. Agrawal and A. El Abbadi. Transaction management in database systems. In [32], chapter 1, pages 1–31. Morgan Kaufmann, 1992.

[5] G. Alonso, S. Blott, A. Fessler, and H.-J. Schek. Correctness and parallelism in composite systems. In *ACM Symposium on Principles of Database Systems*, pages 197–208, Tuscon, AZ, May 12–14 1997.

[6] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Intl. Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[7] F. Bancilhon. Object-oriented database systems. In *ACM Symposium on Principles of Database Systems*, pages 152–162, New York, 1988. ACM.

[8] C. Beeri, P.A. Bernstein, N. Goodman, M.Y. Lai, and D.E. Shasha. A concurrency control theory for nested transactions. In *ACM Symposium on Principles of Database Systems*, pages 45–62, 1983.

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

[10] 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.

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

[12] A.J. Bonner. The power of cooperating transactions. Manuscript, 1997.

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

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

[15] 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`.

[16] A.J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 142–156, Bonn, Germany, September 1996. MIT Press.

[17] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.

[18] A.J. Bonner and M. Kifer. Results on reasoning about action in transaction logic. Submitted for Publication, 1998.

[19] 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.

[20] F. Bry. Intensional updates: Abduction via deduction. In *Intl. Conference on Logic Programming*, Jerusalem, Israel, June 1990.

[21] O. Bukhres and E. Kueshn, Eds. Special issue on software support for work flow management. *Distributed and Parallel Databases—An International Journal*, 3(2), April 1995.

[22] M.A. Casanova. *The Concurrency Control Problem for Database Systems*, volume 116 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1981.

[23] A.K. Chandra and D. Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[24] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[25] W. Chen. Declarative updates of relational databases. *ACM Transactions on Database Systems*, 20(1):42–70, March 1995.

[26] P.K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[27] P.K. Chrysanthis and K. Ramamrithm. ACTA: The SAGA continues. In [32], chapter 10, pages 349–397. 1992.

[28] G. Copeland and D. Maier. Making smalltalk a database system. In *ACM SIGMOD Conference on Management of Data*, pages 316–325, Boston, MA, June 1984.

[29] U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao, and M.-C. Shan. Third generation TP monitors: A database challenge. In *ACM SIGMOD Conference on Management of Data*, pages 393–397, Washington, DD, May 1993.

[30] C. de Maindreville and E. Simon. Non-deterministic queries and updates in deductive databases. In *Intl. Conference on Very Large Data Bases*. Morgan Kaufmann, San Francisco, CA, 1988.

[31] F. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[32] A.K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann, San Mateo, CA, 1992.

[33] A.K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewcz. A multidatabase transaction model for interbase. In *Intl. Conference on Very Large Data Bases*, pages 507–518, Brisbane, Australia, August 13–16 1990.

[34] A. Elmagarmid, Ed. Special issue on unconventional transaction management. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 14(1), June 1991.

[35] J.L. Eppinger, L.B. Mummert, and A.Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, San Mateo, CA, 1991.

[36] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating multi-transaction activities. Technical Report TR-247-90, Princeton University, 1990.

[37] H. Garcia-Molina and K. Salem. Sagas. In *Intl. Conference on Very Large Data Bases*, pages 249–259, May 1987.

[38] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to infrastructure for automation. *Journal on Distributed and Parallel Database Systems*, 3(2):119–153, April 1995.

[39] J. Gray. The transaction concept: Virtues and limitations. In *Intl. Conference on Very Large Data Bases*, pages 144–154, Cannes, France, September 1981.

[40] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[41] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[42] M. Hsu, Ed. Special issue on workflow and extended transaction systems. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 16(2), June 1993.

[43] M. Hsu, Ed. Special issue on workflow systems. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 18(1), March 1995.

[44] Samuel Y.K. Hung. Implementation and Performance of Transaction Logic in Prolog. Master's thesis, Department of Computer Science, University of Toronto, 1996. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.

[45] Setrag Khoshafian and Marek Buckiewicz. *Introduction to Groupware, Workflow, and Workgroup Computing*. John Wiley & Sons, Inc., 1995.

[46] J. Klein. Advanced rule-driven transaction management. In *IEEE COMPCON*. IEEE, 1991.

[47] P.E. Lauer and R.H. Campbell. Formal semantics of a class of high-level primitives for co-ordinating concurrent processes. *Acta Informatica*, 5:297–332, 1975.

[48] B. Liskov. Distributed programming in Argus. *Communications of ACM*, 31(3):300–312, 1988.

[49] N.A. Lynch. Multilevel atomicity: A new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, 1983.

[50] S. Manchanda and D.S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, Los Altos, CA, 1988.

[51] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[52] R. Milner. Operational and algebraic semantics of concurrent processes. In [69], chapter 19, pages 1201–1242. 1990.

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

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

[55] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Series in Information Systems. MIT Press, Cambridge, MA, 1985.

[56] J.G. Mullen and A.K. Elmagarmid. InterSQL: A multidatabase transaction programming language. In *Proceedings of the International Workshop on Database Programming Languages*, pages 399–416, New York City, 30 Aug–1 Sept 1993.

[57] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *ACM Symposium on Principles of Database Systems*, pages 251–262, New York, March 1988. ACM.

[58] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville, MD, 1989.

[59] M. H. Nodine, S. Ramaswamy, and S. B. Zdonik. A cooperative transaction model for design databases. In [32], chapter 3, pages 53–85. 1992.

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

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

[62] G. Phipps, M.A. Derr, and K.A. Ross. Glue-Nail: A deductive database system. In *ACM SIGMOD Conference on Management of Data*, pages 308–317, New York, 1991. ACM.

[63] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond.* Addison-Wesley, 1994.

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

[65] M.P. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proceedings of the International Workshop on Database Programming Languages*, Gubbio, Umbria, Italy, September 6–8 1995.

[66] M.P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proceedings of 12-th IEEE Intl. Conference on Data Engineering*, pages 616–623, New Orleans, LA, February 1996.

[67] L. Stein, S. Rozen, and N. Goodman. Managing laboratory workflow with LabBase. In *Proceedings of the 1994 Conference on Computers in Medicine (CompMed94).* World Scientific Publishing Company, 1995. `ftp://genome.wi.mit.edu/pub/papers/Y1995/workflow.ps.Z`.

[68] Transarc-Encina. *Encina Transactional Processing System: Transactional-C Programmers Guide and Reference, TP-00-D347.* Transarc Corp., Pittsburg, PA, 1991.

[69] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics.* Elsevier, Amsterdam, 1990.

[70] M.Y. Vardi. The complexity of relational query languages. In *Proc. of 14th ACM Symposium on Theory of Computation*, pages 137–146, May 1982.

[71] J. Veijalainen, F. Eliassen, and B. Holtkamp. The s-transaction model. In [32], chapter 12. 1992.

[72] H. Wachter and A. Reuter. The ConTract model. In [32], chapter 7, pages 220–263. 1992.

[73] D.S. Warren. Database Updates in Pure Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 244–253, 1984.

[74] G. Weikum. Principles and realization strategies of multi-level transaction management. Technical Report DVSI-1987-T1, Technical University of Darmstadt, Germany, 1987.

[75] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In [32], chapter 13, pages 515–553. 1992.