Workflow, Transactions and Datalog

Anthony J. Bonner

University of Toronto
Department of Computer Science
Toronto, Ontario, Canada M5S 1A4

`www.cs.toronto.edu/~bonner`
`bonner@db.toronto.edu`

Abstract

Transaction Datalog (abbreviated $\mathcal{TD}$) is a concurrent programming language that provides process modeling, database access, and advanced transactions. This paper illustrates the use of $\mathcal{TD}$ for specifying and simulating workflows, with examples based on the needs of a high-throughput genome laboratory. In addition to traditional database support, these needs include synchronization of work, cooperation between concurrent workflows, and non-serializable access to shared resources. After illustrating workflows, we use $\mathcal{TD}$ to explore the computational properties of workflows in data-intensive applications. We show, for instance, that workflows can be vastly more complex than traditional database transactions, largely because concurrent processes can interact and communicate via the database (*i.e.*, one process can read what another process writes). We then investigate the sources of this complexity, focusing on features for data modeling and process modeling. We show that by carefully controlling these features, the complexity of workflows can be reduced substantially. Finally, we develop a sub-language called *fully bounded* $\mathcal{TD}$ that provides a practical blend of modeling features while minimizing complexity.

A short version of this paper appears in *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems (PODS)*, May 31 - June 2, 1999, Philadelphia, PA, pages 294–305.

Contents

1 Introduction

The management of workflows and business processes is a ubiquitous task faced by many organizations in a wide range of industries [39, 46, 30, 58]. The problem is to coordinate the various activities involved in a complex process, such as trip planning, student registration, telephone installation, laboratory testing, and loan application processing. Because they can access and generate large volumes of data, many workflows require database support, including both data modeling and transaction management. However, because the demands of workflows are more complex than those of traditional database applications, conventional data management techniques are not enough. As one prominent researcher puts it, "The next big challenge for the information technology industry is the management and automation of business processes" [39]. This paper focuses on one aspect of this challenge: high-level languages for specifying workflow. In particular, we investigate the effects of data-oriented and process-oriented features, especially their influence on the computational complexity of workflows.

1.1 Background

The need for database support in workflow management shows up in several ways. For instance, declarative queries and updates are needed both for accessing application data, and for recording and monitoring the history of workflow execution. In fact, this need is so great in some applications that database benchmarks have been developed based on the queries and updates produced by high-volume workflow systems [29, 27]. In addition, since workflows may access shared persistent data, transactions are required. This presents a challenge since workflows can also create dependencies between transactions. For instance, if one transaction fails, then others may have to be started or aborted, depending on the workflow [9, 43, 19]. Such dependencies are not accounted for in the classical theory of transaction management [14, 48]. The result is that "business process management cannot be handled by means of conventional database techniques alone" [39]. For this reason, there has been considerable research on advanced transaction models [43] and their application to transactional workflow [57, 83, 74].

In addition to database support, workflow management requires *process modeling*. A process model describes the flow of control among the various activities that make up a workflow. For instance, in a simple linear workflow, the process model simply lists the order in which the activities must be carried out. In general, a process model can be highly complex, and can specify that some activities be carried out concurrently, that some be synchronized, that some be executed repeatedly, and that some be invoked as subprocesses of others. Numerous formalisms have been developed for this purpose, including various process algebras [68, 55, 53, 13, 10, 69], many kinds of Petri net [73, 63, 64, 41, 7], as well as temporal logic [44, 36], state charts [49], concurrent transition systems [51], and concurrent logic programming [77, 76]. These formalisms all allow concurrently executing processes to interact and communicate, and some (such as process algebras) allow new processes to be created recursively at runtime.

Computational problems associated with workflow have been extensively studied by the process-modeling community. This includes complexity results for problems such as verification, liveness, deadlock detection, and goal reachability (*e.g.*, [51, 35, 12, 44, 36, 65, 71]). However, most of this work focuses on control flow, and either ignores data entirely, or uses a highly simplistic model of data (such as tokens in Petri nets [73]).[1] This leaves many questions unanswered. For instance, what is the influence of data-oriented features (such as queries and updates) on workflow complexity? How does the combination of data modeling and process modeling affect complexity? How does complexity depend on database size? What is the effect of transactional features? This paper begins to address these questions.

Our vehicle for this investigation is *Transaction Datalog* (abbreviated $\mathcal{TD}$), a concurrent programming language that provides both process modeling and database support [21, 19, 15]. $\mathcal{TD}$ has many of the features of process algebras. These include concurrent access to shared resources, communication between sequential processes, and the ability to isolate (or hide) the inner workings of a group of processes from the outside world. Like all process algebras, $\mathcal{TD}$ is compositional, so processes can be defined recursively in terms of subprocesses. It is therefore possible to specify so-called multi-level processes [39], even when the number of levels is determined at runtime. However, unlike process algebras, $\mathcal{TD}$ also provides high-level support for database functions. These include declarative queries, bulk updates, views, and serializability [22, 21, 25]. $\mathcal{TD}$ also has many features of advanced transaction models, including subtransaction hierarchies, relaxed ACID requirements, and fine-grained control over abort and rollback [19]. This integration of process modeling and database functionality is reflected in the formal semantics of $\mathcal{TD}$, which is based on *both* database states *and* events, while the semantics of process algebras is based entirely on events.

1.2 Computational Complexity

As mentioned above, most computational studies of workflow and process models focus on problems such as verification, deadlock detection, goal reachability, etc. Unfortunately, while many of these problems are decidable for finite state machines, they are fully undecidable (outside of RE) for many realistic workflows, where the universe of database states is infinite. To deal with this problem, some database researchers have chosen to restrict the process model instead of the data model [6, 38, 82]. Unfortunately, the result, once again, is that many realistic workflows are not considered.

In this paper, we take a different approach and address a different set of questions. Our goal is to start with a highly expressive workflow language (like $\mathcal{TD}$), and pinpoint the sources of complexity, including the tradeoff between data-oriented and process-oriented features. We do not attempt to reason about undecidable properties of workflows. Instead, we measure what might be called the "data complexity" of a workflow. Specifically, we treat a workflow as a concurrent program that accesses a database. The process starts from an initial database state, updates the database as it executes, and leaves the database in some final state when it terminates (if it terminates). In this way, the workflow defines a partial mapping from initial to final states, just as a Turing machine does. More generally, a non-deterministic workflow defines a binary relation on states. We define the data complexity of a workflow to

[1] An exception is a kind of Petri net known as predicate/transition nets [64]. However, to our knowledge, there has been no analysis of these nets comparable to that presented in this paper.

be the complexity of this relation. With this approach, we can measure the complexity of executing a single workflow instance, or the complexity of spawning many concurrent (and possibly interacting) instances for processing a stream of work items. Sections 3 and 6.1 gives examples of both.

This approach also allows us to compare $\mathcal{TD}$ with other transaction languages, since their complexity has been defined in the same way [5, 1]. The most striking result is that workflows defined in $\mathcal{TD}$ can be much more complex than traditional database transactions. For instance, Section 5.1 shows that $\mathcal{TD}$ can simulate an arbitrary Turing machine, even though it has none of the features that normally lead to this kind of power in a database language. For instance, $\mathcal{TD}$ does not expand the data domain or the database schema during program execution. Database languages with this property are said to be *safe*, and typically their data complexity is within PSPACE [2, 4, 5, 1, 32]. In contrast, the data complexity of $\mathcal{TD}$ is complete for RE. This dramatic increase in complexity, from PSPACE to RE, is due to a property of $\mathcal{TD}$ not supported by traditional database languages: *cooperative concurrency*, *i.e.*, the ability of concurrently executing programs to communicate, synchronize, or otherwise cooperate [50]. Cooperative concurrency is the dominant form of concurrency in distributed systems and process modeling, and is an essential element of workflow [42, 54, 67, 68, 55].[2]

In concurrent programming languages, cooperative concurrency is not a primitive modeling feature, but is the result of combining several more-basic features. In $\mathcal{TD}$, this includes process-oriented features such as sequential and concurrent composition, as well as data-oriented features such as queries and updates. Together, these features allow sequential processes to execute concurrently and to interact via the database, since one process can read what another process writes. Section 5 studies these features and their effect on complexity in detail. We first develop a family of simple syntactic restrictions, where each restriction eliminates a single modeling feature. We then show that these restrictions reduce the complexity of $\mathcal{TD}$ to various levels, including EXPTIME, PSPACE, PTIME and LOGSPACE. These results pinpoint the precise effect on complexity of particular modeling features. However, each feature plays an important role in defining workflows, and in practice, we do not want to entirely eliminate any one of them. With this in mind, Section 6 develops a more complex restriction, called *full boundedness*, that provides a practical blend of modeling capabilities, and is complete for NP. Of course, we would prefer a restriction whose complexity is in PTIME, but cooperative concurrency is inherently non-deterministic, and leads very quickly to NP-completeness.

In general, our complexity results are in line with and improve upon related work in the literature. For instance, like $\mathcal{TD}$, most process algebras can simulate arbitrary Turing machines [80]. Moreover, Harel has shown that cooperative concurrency increases the complexity of many problems by an exponential, even when the number of processes is carefully bounded [51, 52, 50]. Thus, since safe transaction languages are typically PSPACE-complete, one might expect that adding a bounded number of concurrent processes would increase their complexity to EXPSPACE. In this light, our syntactic restrictions are very effective at keeping complexity down.

[2]In contrast, the massive parallelism that is possible in evaluating some database queries is *non-cooperative* concurrency.

Additional papers about $\mathcal{TD}$, a prototype implementation, and the results of benchmark tests are available on the Web at: `http://www.cs.toronto.edu/~bonner/transaction-logic.html`

2 Overview of Transaction Datalog

Transaction Datalog is a fragment of *Concurrent Transaction Logic* (abbreviated $\mathcal{CTR}$), which we developed in previous work [21]. $\mathcal{CTR}$ is an extension of classical logic that seamlessly integrates concurrency and communication with queries and updates. It has a purely logical semantics, including a natural model theory and a sound-and-complete proof theory. Like classical logic, $\mathcal{CTR}$ has a "Horn" fragment with a procedural interpretation, in which programs can be specified and executed. $\mathcal{TD}$ is based on the Horn fragment of $\mathcal{CTR}$, just as classical Datalog is based on the Horn fragment of classical logic. This section reviews the syntax of $\mathcal{TD}$ and its procedural interpretation, summarizing material from [19]. In this discussion, and the rest of the paper, we adopt the terminology of deductive databases.

$\mathcal{TD}$ provides three operators for combining simple programs into more complex ones: sequential composition, denoted $\otimes$; concurrent composition, denoted $|$; and a modality of isolation, denoted $\odot$. In addition, logical rules provide a subroutine facility, exactly as in classical Datalog. The following definition makes the syntax more precise.

Definition 2.1 (Syntax) An atomic formula is a *goal*. If ϕ_1 and ϕ_2 are goals, then so are the formulas $\phi_1 \otimes \phi_2$, $\phi_1 \mid \phi_2$ and $\odot\phi_1$. If ϕ is a goal and p is an atomic formula, then $p \leftarrow \phi$, is a *rule*. A finite set of rules is a *rulebase*. A rulebase together with a goal is a *program*. □

For convenience, we assume that $\otimes$ binds more tightly than $|$. Thus, the expression $p \mid q \otimes r$ is parsed as $p \mid (q \otimes r)$.

Intuitively, goals are procedures, and rules are subroutine definitions (in the logic-programming tradition). In particular, if α and β are goals, then

- $\alpha \otimes \beta$ means "*First execute α, then execute β, and commit iff both α and β commit.*"
- $\alpha \mid \beta$ means "*Execute α and β concurrently, and commit iff both α and β commit.*"
- $\odot\alpha$ means "*Execute α in isolation, and commit iff α commits.*"
- $p \leftarrow \alpha$ means "*An execution of α is also an execution of p, where p commits if α commits.*"

Using these four operators, a $\mathcal{TD}$ programmer combines elementary operations into complex processes. In general, an elementary operation can be any activity that accesses a database, including activities that require human intervention (as in many workflows). Examples include simple database updates, complex application programs, and legacy systems. Semantically, an elementary operation is treated as a black box. This idea is formalized in [21].

In general, the complexity of a $\mathcal{TD}$ program depends on its elementary operations. However, we would like to factor out these operations in order to focus on the complexity of $\mathcal{TD}$ itself. For this reason, we base our complexity analysis on a small set of simple operations. This does not result in any loss of expressiveness or computational power, since with these operations, $\mathcal{TD}$ is expressively complete, *i.e.*, it can express any computable database transaction (Theorem 5.5). These operations are also minimal, since if any one of them is removed, then expressive completeness is lost.[3]

In this paper, we use four kinds of elementary operation, which we denote by four kinds of atomic formula: $p(\overline{x})$, $p.empty$, $ins.p(\overline{x})$, $del.p(\overline{x})$, where p is a base predicate symbol. The formal semantics of these formulas is given in [21, 19]. Intuitively, the first two formulas are yes/no queries, and the last two formulas are updates. In particular,

- $p(\overline{x})$ means "*Commit iff $p(\overline{x})$ is in the database.*"
- $p.empty$ means "*Commit iff the database contains no atoms of the form $p(\overline{x})$.*"
- $ins.p(\overline{x})$ means "*Insert atom $p(\overline{x})$ into the database, and commit.*"
- $del.p(\overline{x})$ means "*Delete atom $p(\overline{x})$ from the database, and commit.*"

These elementary operations can be combined into more-complex database programs. For instance, the goal $del.p(a) \otimes del.p(b)$ is a simple program that first deletes $p(a)$ from the database and then deletes $p(b)$. Likewise, the goal $[del.p(a) \otimes del.p(b)] \mid [ins.q(a) \otimes ins.q(b)]$ is a program consisting of two sequential processes that execute concurrently, where one process deletes $p(a)$ and $p(b)$, while the other process inserts $q(a)$ and $q(b)$. Finally, the rule $r(X) \leftarrow del.p(X) \otimes ins.q(X)$ defines a subroutine with name r and parameter X. If the goal $r(b)$ is executed, then the subroutine is invoked with b as the parameter value, in which case $p(b)$ is first deleted from the database, and $q(b)$ is then inserted.

2.1 Isolation and Transactions

A program executes in isolation if it does not communicate or interact with other programs. Isolation is a fundamental property of database transactions [14, 48], and is closely related to serializability. For instance, if t_1, t_2, ..., t_n are database programs, then the goal $\odot t_1 \mid \odot t_2 \mid \cdots \mid \odot t_n$ executes them serializably. In $\mathcal{TD}$, the modality of isolation supports a wide variety of important database functions. We briefly describe a few of these here.

Transaction Programs. $\mathcal{TD}$ can be used to program database transactions. For example, the goal $\odot[del.p(a) \otimes del.p(b)]$ is a simple transaction program. Such programs execute in isolation, and they either commit or abort. This particular program always commits since its components, $del.p(a)$ and $del.p(b)$, always commit. As another example, the goal

[3] Another approach would be to prove *relativized* complexity results [72, 11] with elementary operations modeled by a variation of oracle Turing machines. Most of the results in this paper can be relativized in this way.

$\odot[p(b) \otimes del.p(b)]$ represents a transaction program with a precondition, $p(b)$. This program first asks if $p(b)$ is in the database, and if so, it deletes $p(b)$. This program commits if $p(b)$ is in the database at the start of execution, and aborts otherwise.[4] By using rules, a programmer can define transactional subroutines. For instance, the rule $r(X) \leftarrow \odot[p(X) \otimes del.p(X)]$ defines a transaction with name r and parameter X. Using b as the parameter value, $r(b)$ commits if $p(b)$ is in the database at the start of execution.

Advanced Transactions. As shown in [19], $\mathcal{TD}$ accounts for many basic properties of "advanced" transaction models [43], including nested transactions. These properties include subtransaction hierarchies, non-vital subtransactions, relative commit, and partial rollback. For example, the goal $\odot(\phi_1 \mid \odot(\phi_2 \mid \odot\phi_3))$ is a transaction, which contains a subtransaction, $\odot(\phi_2 \mid \odot\phi_3)$, which contains a sub-subtransaction, $\odot\phi_3$. As another example, in the transaction $\odot(\phi_1 \otimes \phi_2)$, the commit of ϕ_1 is *relative* to the whole transaction: if ϕ_2 aborts, then the whole transaction aborts and is undone; in particular, ϕ_1 is undone, even though it has already terminated and committed.

Declarative Queries. When an isolated $\mathcal{TD}$ program contains no updates, it defines a read-only transaction, *i.e.*, a query. As shown in Section 4.2, for such programs, the connectives $\otimes$ and $\mid$ both reduce to classical conjunction, and Transaction Datalog reduces to classical Datalog. For instance, the goals $\odot[p(X,Y) \otimes q(Y,Z)]$ and $\odot[p(X,Y) \mid q(Y,Z)]$ both express the join of relations p and q. Any classical Datalog query can be expressed in this fashion. For instance, given the following two rules:

$$tc(X,Y) \leftarrow p(X,Y) \qquad\qquad tc(X,Y) \leftarrow p(X,Z) \mid tc(Z,Y)$$

the goal $\odot\, tc(X,Y)$ expresses the transitive closure of predicate p. Transaction Datalog is therefore an extension of classical Datalog.

Example 2.2 (Nested Transaction Programs) Suppose the balance of a bank account is given by the relation *balance*$(Acct, Amt)$. The rules below define four transaction programs: *change*$(Acct, Bal_1, Bal_2)$, which changes the balance of account $Acct$ from Bal_1 to Bal_2; *withdraw*$(Amt, Acct)$, which withdraws an amount from an account; *deposit*$(Amt, Acct)$, which deposits an amount into an account; and *transfer*$(Amt, Acct_1, Acct_2)$, which transfers an amount from account $Acct_1$ to account $Acct_2$. The programs *transfer*, *withdraw* and *deposit* are defined as transactions, while *change* is simply a subroutine. In addition, *withdraw* and *deposit* are nested within *transfer*.

$$\begin{array}{l}
transfer(Amt, Acct_1, Acct_2) \;\leftarrow\; \odot[withdraw(Amt, Acct_1) \mid deposit(Amt, Acct_2)] \\
withdraw(Amt, Acct) \;\leftarrow \\
\qquad\qquad \odot[balance(Acct, Bal) \otimes Bal > Amt \otimes change(Acct, Bal, Bal - Amt)] \\
deposit(Amt, Acct) \;\leftarrow\; \odot[balance(Acct, Bal) \otimes change(Acct, Bal, Bal + Amt)] \\
change(Acct, Bal_1, Bal_2) \;\leftarrow\; del.balance(Acct, Bal_1) \otimes ins.balance(Acct, Bal_2)
\end{array}$$

[4] In contrast, in a *non-isolated* execution of $p(b) \otimes del.b(b)$, if $p(b)$ were *not* in the database, then the execution would *wait* for some other program to put it there. This idea is illustrated in Examples 3.3 and 3.4.

The first rule says, to transfer amount Amt from $Acct_1$ to $Acct_2$, withdraw Amt from $Acct_1$, and concurrently deposit Amt in $Acct_2$. The second rule says, to withdraw an amount from an account, first retrieve the account balance, then check that it is big enough, then change the balance from Bal to $Bal - Amt$. The last rule changes the balance of an account by deleting the old balance and inserting the new one. The *transfer* transaction aborts if either subtransaction, *withdraw* or *deposit*, aborts. Likewise, *withdraw* aborts if the query $balance(Acct, Bal)$ fails (*e.g.*, if $Acct$ is not a valid account) or if the test $Bal > Amt$ fails (*i.e.*, if the balance is too small). □

Example 2.2 is a canonical example of nested transactions, one that brings out several limitations of the classical (or "flat") transaction model [47, 70, 48, 43, 19]. 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 completed its execution and 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, transaction combinations can now behave like individual transactions. 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 possibilities are not supported by the classical transaction model. Further discussion of nested transactions, along with examples specified in $\mathcal{TD}$ can be found in [19].

3 Examples of Workflow

Like classical logic programs, $\mathcal{TD}$ programs have both a declarative semantics and a procedural interpretation. These have been published in detail elsewhere [21, 19, 22], and are reviewed in Appendices A and B. This section illustrates the procedural interpretation through a series of examples on workflow. Each example has been tested on our prototype implementation [60, 78], and performs exactly as described below.

The examples focus on so-called *production* workflow, which forms the core of a business or enterprise [46, 66]. Production workflows are typically complex, well defined, high volume, and mission critical. 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. Because they process large numbers of work items, production workflows are often data intensive. A concrete example is the laboratory workflows used at the Whitehead Institute/MIT Center for Genome Research [28, 29, 79], which is engaged in several large-scale genome mapping and sequencing projects [37]. Each project involves the completion of tens of millions of experiments, organized into a network of factory-like production lines. Coordinating the flow of materials through the production lines, and recording and querying the history of experimental steps and the results they produce are the main data and workflow management requirements [28]. The workflows at the Genome Center are data intensive. In fact, as laboratory automation increased, database performance became a bottleneck in workflow throughput, and we had to develop a workflow/database benchmark to evaluate new storage managers for their laboratory information system [29, 27, 28]. To make the examples in this paper more concrete, we shall sometimes describe them in terms of genome laboratory workflow.

Example 3.1 (Workflow Specification) The three rules below define a simple workflow made up of a collection of tasks and a sub-workflow. The tasks execute sequentially and concurrently, and the flow of work in the sub-workflow depends on a database query. Intuitively, the predicate $workflow(W)$ represents the flow of work for a single work item, W. Likewise, the predicate $subflow(W)$ represents the sub-workflow, and the predicate $task_i(W)$ represents a workflow task.

$$\begin{aligned} workflow(W) &\leftarrow task_1(W) \otimes [task_{2a}(W) \mid task_{2b}(W)] \otimes task_3(W) \otimes subflow(W) \\ subflow(W) &\leftarrow p(W) \otimes task_4(W) \otimes task_5(W) \\ subflow(W) &\leftarrow q(W) \otimes task_6(W) \otimes task_7(W) \end{aligned}$$

The first rule says the following: first $task_1$ should be applied to W; then $task_{2a}$ and $task_{2b}$ should be applied concurrently; then $task_3$ should be applied; and finally W should be passed to a sub-workflow for further processing. The sub-workflow applies a series of tasks to W depending on the outcome of a test: if $p(W)$ is true, then $task_4$ and $task_5$ are applied; but if $q(W)$ is true, then $task_6$ and $task_7$ are applied.[5] Note that p and q are arbitrary database queries, and may be defined by a complex set of $\mathcal{TD}$ rules. Also note the synchronization implicit in the first rule, since $task_3$ cannot start until both $task_{2a}$ and $task_{2b}$ have finished. $\square$

Example 3.2 (Workflow Simulation) The rules below simulate the execution of a workflow on a set of work items. They assume that the predicate $workflow(W)$ has been defined, as in Example 3.1. They also assume that information describing the work items is stored in the database, and can be accessed by the workflow as it executes. In particular, we assume that an identifier for each work item is initially stored in the base predicate $in(W)$, which acts as an "in basket" for the workflow. As the simulation progresses, work items are removed from the in basket, processed by the workflow, and then put into an out basket. The entire process simulates the pipelining found in many workflow systems: items enter the workflow sequentially, but are processed by the workflow concurrently.

$$\begin{aligned} simulate &\leftarrow getItem(W) \otimes [simulate \mid workflow(W) \otimes putItem(W)] \\ simulate &\leftarrow in.empty \\ getItem(W) &\leftarrow \odot[in(W) \otimes del.in(W)] \\ putItem(W) &\leftarrow ins.out(W) \end{aligned}$$

The first rule recursively removes work items from the in basket, and creates a workflow instance for each of them, where all the workflow instances execute concurrently. At each recursive call, the rule removes a work item from the in basket, and then spawns two processes. One process creates a workflow instance for the work item, and when it terminates, puts the work item in the out basket. The other process calls the simulation recursively. The second rule terminates the recursion when there are no work items left in the in basket.

[5]If both tests are true, then the sub-workflow chooses one series of tasks non-deterministically. If neither test is true, then the sub-workflow waits until one of them becomes true.

The third rule retrieves a work item, W, from the in basket, and then deletes it from the basket. Note the use of the modality of isolation, $\odot$, in the rule body. This ensures that the retrieval and deletion are carried out as a single transaction, *i.e.*, as if $getItem(W)$ were an elementary operation. In this way, if two processes both execute $getItem$ concurrently, they will not get the same item. Finally, the fourth rule puts a work item in the out basket, by simply inserting it into the base predicate $out(W)$. □

3.1 Advanced Features

The rulebase in Example 3.2 illustrates all the *basic* features of $\mathcal{TD}$. It uses all four operators ($\otimes$, $|$, $\odot$ and $\leftarrow$), as well as both kinds of elementary update (tuple insertion and tuple deletion) and both kinds of elementary query (tuple testing and emptiness testing). Perhaps surprizingly, these simple features are enough to make $\mathcal{TD}$ a highly expressive language. By using them in combination, $\mathcal{TD}$ can express a variety of sophisticated workflows and a wide range of database functions. For instance, Example 2.2 shows how to express nested transactions, something that few database languages can do. More generally, as shown in [19], $\mathcal{TD}$ can express so-called flexible or non-standard transactions, in which the ACID properties are relaxed [43]. In the rest of this section, we build on Examples 3.1 and 3.2 to show how to express advanced workflow features in $\mathcal{TD}$, including shared resources, communication, synchronization, and active rules. A final example, based on cooperating workflow networks, is given in Section 6.1.

Examples 3.1 and 3.2 can be refined in several ways. For instance, we may wish to process not a *set* of work items, but a *stream* of items that arrive over a period of time (*e.g.*, as new students apply for registration, or new DNA samples arrive for laboratory testing). In this case, the environment provides the workflow with new work items. As is commonly done in process algebras [68, 55], we can treat the environment simply as another process (possibly non-deterministic). That is, instead of executing the goal $simulate$ in Example 3.2, we could execute the goal $simulate \mid environment$, where $environment$ is a process that inserts new work items into the database. For instance, the process

$$ins.in(w_1) \otimes ins.in(w_2) \otimes \cdots \otimes ins.in(w_n)$$

inserts a sequence of n work items, $w_1, w_2, ..., w_n$, into the in basket, one item at a time. In this way, the in basket stores a "pool" of work items that is consumed by the workflow and replenished by the environment. Of course, more-complex environments can also be modeled. In Example 6.4, for instance, two workflows generate multiple input streams for a third workflow.

The examples above can also be refined to account for the sharing of limited resources during workflow execution. To appreciate the problem, observe that Example 3.2 executes concurrently many instances of the workflow in Example 3.1. Normally, however, there are physical limits on the number of workflow instances that may be active at one time. Typically, each task in a workflow is performed by an "agent," (*e.g.*, a machine or a person), only a fixed number of agents is available, and only qualified agents can be assigned to each task [46]. In effect, the agents are resources that must be shared by the various workflow instances, thus limiting the number of instances that can be active at one time. For this reason, an important part of workflow specification is assigning agents to tasks [46]. This is easily accomplished in a language like $\mathcal{TD}$ that integrates databases and processes, since we can record the status of agents in the database, and update them as the workflow progresses. The next example shows one way of doing this. The example also suggests how to keep track of work that has been performed. This allows for monitoring, tracking and querying the status of workflow activities, another important aspect of workflow management [40, 46, 29].

Example 3.3 (Shared Resources) The rules below refine the predicate $task_i(W)$ used in Example 3.1, to account for the resources needed to execute the task. In this case, the resources are qualified agents. The rules assign an agent, A, to carry out $task_i$ on work item W. The rules access two base predicates: $qualified_i(A)$, which means that agent A is qualified to carry out $task_i$; and $available(A)$, which means that agent A is currently not working on any other task. In addition, after agent A has completed the task, the atom $done_i(A, W)$ is inserted into the database, which provides a record of the work performed.

$$task_i(W) \leftarrow request_i(A) \otimes doTask_i(A, W) \otimes release(A) \otimes ins.done_i(A, W)$$

$$request_i(A) \leftarrow \odot[qualified_i(A) \otimes available(A) \otimes del.available(A)]$$

$$release(A) \leftarrow ins.available(A)$$

The first rule requests an agent, A, qualified to perform $task_i$. When such an agent becomes available, the rule invokes the predicate $doTask_i(A, W)$, which represents agent A carrying out $task_i$ on item W. When the task has been completed, the rule releases the agent from service, and inserts a record of the work into the database. The second rule selects the agent requested by the first rule. It retrieves an agent, A, who is qualified to perform $task_i$ and who is available. (If no such agent is currently available, then the process waits until one becomes available.) The rule then deletes A from the pool of available agents. The third rule simply returns A to the pool. □

In many applications, a workflow is made up of sub-workflows that run concurrently and synchronize themselves. This is often the case when a work item is made up of several parts, where each part is processed by a different sub-workflow. Since the parts are related, the workflows may have dependencies between them. Typically, one workflow needs information produced by another workflow, and may have to wait for this information to become available before it can continue. This is the case, for instance, in the workflow described in [29], in which the work items are DNA samples, and the purpose of the workflow is to construct a physical genome map.[6] The next example provides a simple illustration of interacting workflows. A more-complex example is given in Section 6.1.

Example 3.4 (Communication and Synchronization) The rules below specify a workflow with two sub-workflows that execute concurrently. Each sub-workflow executes a sequence of tasks. Moreover, the two sub-workflows are not independent: each contains a task that cannot execute until the other sub-workflow reaches a certain point.

[6]This particular workflow consists of two concurrent sub-workflows that synchronize themselves at several points. One sub-workflow processes DNA samples called *clones*, and the other processes shorter samples called *tclones*. At several points, the clone workflow needs information generated by the tclone workflow.

$$
\begin{aligned}
workflow(W) &\leftarrow subflow_1(W) \mid subflow_2(W) \\
subflow_1(W) &\leftarrow task_{1a}(W) \otimes task_{1b}(W) \otimes done_{2b}(W) \otimes task_{1c}(W) \\
subflow_2(W) &\leftarrow task_{2a}(W) \otimes done_{1a}(W) \otimes task_{2b}(W) \otimes task_{2c}(W)
\end{aligned}
$$

The first rule simply splits the main workflow into two sub-workflows, denoted $subflow_1$ and $subflow_2$, each consisting of three tasks. As in Example 3.3, we assume that upon completion, each task inserts an atom into the database, as a record of its activity. Specifically, we assume that $task_i(W)$ inserts the atom $done_i(W)$. Observe that both sub-workflows contain atoms of the form $done_i(W)$. When these atoms appear in a goal (as they do here), they represent transactions that cannot commit until the atom is true. Thus, $done_i(W)$ will not commit until $task_i(W)$ has finished. The execution of the workflow is therefore subject to two constraints. First, $task_{1a}(W)$ must finish before $task_{2b}(W)$ can start. This is because $subflow_2$ will wait at the atom $done_{1a}(W)$ until it is true. Likewise, $task_{2b}(W)$ must finish before $task_{1c}(W)$ can start. This is because $subflow_1$ will wait at the atom $done_{2b}(W)$ until it is true. In effect, inserting the atom $done_i(W)$ into the database sends a synchronization message from one sub-workflow to the other. □

In Example 3.1, the flow of work is fairly mechanical and could easily be described by a flowchart. This is typical of many production workflows, in which the flow of work may be dictated by the physical structure of factory-like production lines. In other applications, however, the flow of work may be much more dynamic and varied. For instance, for a particular work item, the flow of work may not be determined until run time, and different work items may be processed in vastly different ways. This is already hinted at in Example 3.1, where the path taken through the sub-workflow is determined at runtime by the result of a database query. If a workflow involves a large number of conditions that must continually be tested, then it may be convenient to describe the work in terms of condition-action rules, in which a task (or a sub-workflow) is triggered when a given condition arises. This is typical of some applications in electronic commerce, as described in [59]. For instance, when a user interacts with a web-based store front, the tasks performed by the store-front (such as popping up promotions on the screen) may depend on the user's prior history with the store, and on what the application learns about the user as the session progresses [59]. The next example illustrates how such workflows can be specified in $\mathcal{TD}$.

Example 3.5 (Condition-Action Rules) The rulebase below specifies a workflow in which a process directed by a user is monitored by a set of condition-action rules. Intuitively, the predicate $workflow(U)$ represents the flow of work for a particular user, U, the predicate $user(U)$ represents the user's process, and the predicate $triggers(U)$ represents the set of condition-action rules. Each condition-action rule causes an action to execute when a given condition arises.

$$
\begin{aligned}
workflow(U) &\leftarrow init(U) \otimes [triggers(U) \mid user(U) \otimes ins.done(U)] \otimes cleanup(U) \\
triggers(U) &\leftarrow trigger_1(U) \mid trigger_2(U) \mid \cdots \mid trigger_n(U) \\
trigger_i(U) &\leftarrow condition_i(U) \otimes action_i(U) \\
trigger_i(U) &\leftarrow done(U)
\end{aligned}
$$

The first rule specifies a simple workflow with three phases: first, execute an initialization procedure; then, activate the condition-action rules and execute the user's process; and finally, execute a cleanup procedure. The condition-action rules are activated when the predicate $triggers(U)$ is executed. This predicate executes concurrently with the user process, and immediately invokes all the condition-action rules. Notice that when the user's process is finished, the atom $done(U)$ is inserted into the database. This effectively broadcasts a message that deactivates any condition-action rules that did not fire. Each condition-action rule (or trigger) is represented by a predicate of the form $trigger_i$, which is defined by two $\mathcal{TD}$ rules. The first rule says to wait until $condition_i$ becomes true, and then execute $action_i$. The second rule says to deactivate the trigger when the user's process is done. For each i, exactly one of these two rules will execute. Thus, either $action_i$ is executed, or the trigger is eventually deactivated. □

It is worth noting that the triggers in Example 3.5 can be viewed not just as condition-action rules, but as as event-condition-action rules in the active-database tradition []. In this case, all the rules are activated by a common event: the creation and intialization of a workflow instance. More generally, different sets of triggers can be activated at different points in a workflow. As a simple example, the following goal is a sequence of two sub-workflows, each activating a different set of triggers:

$$[subflowA \mid triggersA] \otimes [subflowB \mid triggersB]$$

Here, $triggersA$ is a set of condition-action rules that are activated during the execution of $subflowA$. Likewise, $triggersB$ is activated during $subflowB$. In this way, $\mathcal{TD}$ allows reactive workflows based on condition-action rules to be seamlessly integrated with structured workflows based on flow charts. Moreover, by embedding active rules inside a structured workflow, we effectively endow them with modularity, a feature notoriously lacking from the flat and unstructured world of active databases.

4 Executional Entailment

This section introduces the notion of *executional entailment*, which describes the effect of executing a $\mathcal{TD}$ program. In this paper, we use executional entailment in several ways: to describe the transactional properties of $\mathcal{TD}$ programs, to show that Transaction Datalog is an extension of classical Datalog, and (in Section 5) to define the data complexity of $\mathcal{TD}$.

Recall that a $\mathcal{TD}$ program is defined by a rulebase, $\mathbf{P}$, and a goal, ϕ. An executional entailment for this program is an expression of the form $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$. Intuitively, this expression means that when the program is executed in isolation, it can transform database $\mathbf{D}_1$ into database $\mathbf{D}_2$. For example, for any rulebase $\mathbf{P}$,

$$\mathbf{P}, \{ab\}\,\{\,\} \;\models\; del.a \otimes del.b \qquad\qquad \mathbf{P}, \{\,\}\,\{cd\} \;\models\; ins.c \otimes ins.d$$

$$\mathbf{P}, \{ab\}\,\{cd\} \;\models\; (del.a \otimes del.b) \mid (ins.c \otimes ins.d)$$

Likewise, if $\mathbf{P}$ contains the two rules $p \leftarrow del.a \otimes del.b$ and $q \leftarrow ins.c \otimes ins.d$, then

$$\mathbf{P}, \{ab\}\,\{\,\} \models p \qquad\qquad \mathbf{P}, \{\,\}\,\{cd\} \models q$$

$$\mathbf{P}, \{ab\}\,\{cd\} \models p \mid q$$

We say that a program *succeeds* from database $\mathbf{D}_1$ if $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$ for some database $\mathbf{D}_2$. Otherwise, the program *fails* from $\mathbf{D}_1$. For example, the goal $b \otimes del.b$ succeeds if and only if the database initially contains the atom b; *i.e.*, this goal successfully deletes b from the database if b is there to begin with, and fails otherwise. The formal semantics of executional entailment is given in [21, 19], and is reviewed in detail in Appendices A and B. This semantics integrates queries, updates and processes in a simple logical framework.

4.1 Transactional Properties

This section uses executional entailment to illustrate some basic properties of $\mathcal{TD}$ programs. These properties are all transactional since executional entailment describes the effects of *isolated* executions, *i.e.*, executions that involve no interaction with the outside world.[7] Of course, the programs themselves may have a rich internal structure of interacting concurrent processes, but this is invisible to the outside world (except for its effect on complexity, as shown in Section 5).

We begin with elementary database operations. Recall that the version of $\mathcal{TD}$ used in this paper has four kinds of elementary operation: tuple testing, $p(x)$; tuple insertion, $ins.p(x)$; tuple deletion, $del.p(x)$; and emptiness testing, $p.empty$. (Here, p is a base predicate symbol, and thus does not appear in any rule heads.) These operations have the following formal properties:

1. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models ins.p(\overline{x})$ *iff* $\mathbf{D}_2 = \mathbf{D}_1 \cup \{p(\overline{x})\}$
2. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models del.p(\overline{x})$ *iff* $\mathbf{D}_2 = \mathbf{D}_1 - \{p(\overline{x})\}$
3. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models p(\overline{x})$ *iff* $\mathbf{D}_1 = \mathbf{D}_2$ *and* $p(\overline{x}) \in \mathbf{D}_1$
4. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models p.empty$ *iff* $\mathbf{D}_1 = \mathbf{D}_2$ *and* $p(\overline{x}) \notin \mathbf{D}_1$ *for all* $\overline{x}$.

These properties have a natural interpretation in terms of the operational semantics of $\mathcal{TD}$. For instance, Property 1 says that the transaction $ins.p(\overline{x})$ transforms database $\mathbf{D}_1$ into database $\mathbf{D}_1 \cup \{p(\overline{x})\}$, and that it always succeeds (*i.e.*, commits). In contrast, Property 3 says that the transaction $p(\overline{x})$ has no effect on the database, but that it succeeds if and only if the atom $p(\overline{x})$ is in the database.

We can also state some basic properties of compound transactions, *i.e.*, transactions defined using the logical connectives of $\mathcal{TD}$. For instance,

5. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_3 \models \alpha \otimes \beta$ *iff* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \alpha$ *and* $\mathbf{P}, \mathbf{D}_2\mathbf{D}_3 \models \beta$ *for some state* $\mathbf{D}_2$.
6. *If* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$ *then* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models q$
   where $q \leftarrow \phi$ *is a ground instantiation of a rule in* $\mathbf{P}$.

[7] The effects of *non-isolated* execution have a more complex description involving execution traces, as described in Appendix A.

7. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \odot\alpha \mid \odot\beta$ *iff* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \alpha \otimes \beta$ *or* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \beta \otimes \alpha$

8. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$ *iff* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \odot\phi$

Again, these properties all have a natural interpretation. For instance, Property 7 says that when isolated programs execute concurrently, the execution is serializable. Property 8 says simply that the expression $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$ refers only to *isolated* executions of ϕ.

We now use these properties to illustrate a basic feature of $\mathcal{TD}$.

4.2 Classical Datalog $\subset$ Transaction Datalog

Transaction Datalog is an extension of Classical Datalog. In particular, classical Datalog corresponds to the fragment of $\mathcal{TD}$ in which tuple testing is the only elementary operation. To see this, first note that since database updates are not allowed, this fragment of $\mathcal{TD}$ defines read-only transactions, *i.e.*, queries. Moreover, any program in this fragment can be trivially transformed into classical Datalog: simply remove all modalities of isolation, and replace sequential and concurrent composition by classical conjunction. For instance, the rule $b \leftarrow (c_1 \otimes c_2) \mid \odot(c_3 \otimes c_4)$ in Transaction Datalog is transformed into the rule $b \leftarrow c_1 \wedge c_2 \wedge c_3 \wedge c_4$ in classical Datalog. Clearly, any classical Datalog program can be generated in this way, as illustrated in Section 2.1. The following theorem shows that the transformed program and the original $\mathcal{TD}$ program are equivalent, *i.e.*, they express the same query.

Theorem 4.1 (Relationship to Classical Datalog) *Let* $\mathbf{P}$ *be a rulebase, let* ϕ *be a goal, and let* $\mathbf{P}^c$ *and* ϕ^c *be their transformed versions as described above. If tuple tests are the only elementary operations in* $\mathbf{P}$ *and* ϕ*, then* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$ *if and only if* $\mathbf{D}_1 = \mathbf{D}_2$ *and* $\mathbf{P}^c \cup \mathbf{D}_1 \models^c \phi^c$*, for any pair of databases,* $\mathbf{D}_1$*,* $\mathbf{D}_2$*, where* $\models^c$ *denotes entailment in classical logic.*

We briefly indicate why this theorem is true by appealing to the eight properties described above. First note that the the theorem clearly holds for elementary queries, *i.e.*, when ϕ is an atomic formula, $p(x)$, and p is a base predicate symbol. In this case, $\phi = \phi^c$, and the theorem follows immediately from Property 3. That is, $\mathbf{P}, \mathbf{DD} \models p(x)$ *iff* $p(x) \in \mathbf{D}$ *iff* $\mathbf{P}^c \cup \mathbf{D} \models^c p(x)$, since p does not appear in the head of any rule in $\mathbf{P}$ or $\mathbf{P}^c$. Second, in the absense of updates, we can show that the connectives $\otimes$ and $\mid$ both behave like classical conjunction. This is because the database state does not change during program execution, so the order of operations does not affect the outcome. For instance, without updates, Property 5 reduces to the following:

$$\mathbf{P}, \mathbf{DD} \models \alpha \otimes \beta \quad \textit{iff} \quad \mathbf{P}, \mathbf{DD} \models \alpha \quad \textit{and} \quad \mathbf{P}, \mathbf{DD} \models \beta$$

It is not hard to show that a similar property holds for concurrent composition. Third, in the absense of updates, rules in $\mathcal{TD}$ behave like rules of classical Datalog. In particular, Property 6 reduces to the following:

$$\textit{if} \quad \mathbf{P}, \mathbf{DD} \models \phi \quad \textit{then} \quad \mathbf{P}, \mathbf{DD} \models q$$

where $q \leftarrow \phi$ is a ground instantiation of a rule in $\mathbf{P}$. Finally, we note that in the absense of updates, all executions are serializable and thus isolated, so the modality of isolation is simply not needed. These properties indicate why Theorem 4.1 is true. A complete proof will soon be available in the appendices.

Theorem 4.1 suggests a natural optimization. Suppose that a $\mathcal{TD}$ goal (or rule body) contains a formula of the form $\odot p(x)$, where p is a predicate defined by rules in which tuple testing is the only elementary operation. The formula $\odot p(x)$ is therefore a query, and according to Theorem 4.1, it can be treated as a query of classical Datalog. This means that workflows defined in $\mathcal{TD}$ can contain arbitrary Datalog queries, and that well-known optimization techniques (such as magic sets or tabling) can be applied to them.

5 Data Complexity

The rest of this paper establishes the data complexity of $\mathcal{TD}$, and investigates the sources of this complexity, focusing first on data-oriented features, and then on process-oriented features. (The next section considers the interplay of the two.) As discussed in Section 1, we measure the complexity of a $\mathcal{TD}$ program by treating it as a transaction and observing its effect on the database. This effect is formalized by the notion of executional entailment introduced in Section 4.

Our complexity results are based on a number of standard definitions, adapted from [4, 5, 31]. A database schema is a finite set of base predicate symbols (with associated arities). A database with schema S is a finite set of ground atoms constructed from the predicate symbols in S. The domain of a database is the set of constant symbols appearing in it. The domain of database $\mathbf{D}$ is denoted $dom(\mathbf{D})$. A database transaction of type $\langle S_1, S_2 \rangle$ is a binary relation on databases of schema S_1 and S_2. A transaction T is safe if $dom(\mathbf{D}_2) \subseteq dom(\mathbf{D}_1)$ for every pair $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$ in T. Consider a $\mathcal{TD}$ program defined by rulebase $\mathbf{P}$ and goal ϕ. The program *expresses* a transaction T of type $\langle S_1, S_2 \rangle$ if for any database $\mathbf{D}_1$ with schema S_1, the pair $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle \in T$ if and only if $\mathbf{P}, \mathbf{D}_1 \mathbf{D}_2 \models \phi$. The data complexity of a transaction, T, is the complexity of recognizing the elements of T, that is, of determining whether a given database pair is in T. The data complexity of $\mathcal{TD}$ is the complexity of the most complex transaction expressed by a $\mathcal{TD}$ program. Likewise for the data complexity of a subset of $\mathcal{TD}$. For instance, given a subset of $\mathcal{TD}$, if the most complex transaction is an NP-complete language, then we say that this subset is *data complete* for NP.

Proofs of the theorems below are given in the appendices.

5.1 Data-Oriented Features

This section shows how the complexity of $\mathcal{TD}$ depends on the elementary database operations. We first show that updates have a huge effect on the computational power of $\mathcal{TD}$, far beyond that of other database languages. We then show that both insertions and deletions are needed to achieve this power. Finally, we show that with all four elementary operations, $\mathcal{TD}$ is an expressively-complete transaction language.

We first consider the special case in which tuple-testing is the only elementary operation. In this case, $\mathcal{TD}$ is a query language, and as shown in Section 4.2, it reduces to classical Datalog. Since the data complexity of classical Datalog is complete for PTIME, we have the following result:

Corollary 5.1 *With tuple testing as the only elementary operation, $\mathcal{TD}$ is data complete for PTIME.*

This corollary provides a sharp contrast to our next result, which accomodates updates. Specifically, we show that when updates are added to $\mathcal{TD}$, its data complexity skyrockets from PTIME to RE. From a database perspective, this result is the most unusual in the paper, since $\mathcal{TD}$ has none of the features that normally lead to RE-completeness in a database language. In particular, it is a *safe* language that does not generate an unbounded number of tuples during transaction execution. Normally, the data complexity of such languages is confined to PSPACE [4, 5, 1, 32]. The vastly greater power of $\mathcal{TD}$ comes from an ability that is essential to workflow, but is lacking in other update languages: interacting concurrent processes.

Theorem 5.2 *With tuple testing, tuple insertion, and tuple deletion as the only elementary operations, $\mathcal{TD}$ is data complete for RE.*

In contrast to $\mathcal{TD}$, many transaction languages achieve RE-completeness by abandoning safety. Often, they expand the data domain during transaction execution [4, 5, 2, 26] or they expand the database schema [31, 2]. Typically, to prove RE-completeness, the tape of a Turing machine is encoded as a database, and the finite control is encoded as a database program. In such approaches, database queries simulate tape reads, and database updates simulate tape writes. The database then grows to arbitrary size during program execution, since it encodes a machine tape that grows to arbitrary length.

We take a completely different approach. To prove the RE-completeness of $\mathcal{TD}$, we encode *both* the finite control *and* the machine tape as database programs. These programs execute concurrently, and use *communication* to simulate tape operations. In particular, to read from a tape, the control program sends a "read message" to the tape program; and to write, it sends a "write message." Here, the database is the medium of communication: to send a message between programs, one program reads what the other one writes. In this way, we use queries and updates to simulate communication, as illustrated in Example 3.4. The result is that $\mathcal{TD}$ achieves RE-completeness with a *fixed* data domain, and a *fixed* database schema, and thus with databases of *polynomial* size.

Our encoding of Turing machines in $\mathcal{TD}$ has some resemblance to the encoding of Turing machines in process algebras such as CCS [80]. However, in CCS, the simulation relies heavily on the *restriction operator* [80, 35], which allows a CCS program to create an unbounded number of new (private) communication channels during execution. Since $\mathcal{TD}$ is built around databases, not communication channels, restriction is not a primitive operation in $\mathcal{TD}$. Instead, our simulation relies on sequential composition (which is not a primitive operation in CCS). In effect, to encode a tape storing the string *abcd*, we use the sequential program $a \otimes b \otimes c \otimes d$. Of course, this is not the whole story, since in addition to encoding the tape contents, we must be able to read and write the tape as well. For this, updates and concurrency are essential. In fact, if either of these elements is removed from $\mathcal{TD}$, then its computational power drops dramatically, as shown in Corollary 5.1 and Theorem 5.6.

The proof of Theorem 5.2 uses insertion and deletion to simulate communication. However, Example 3.4 shows that concurrent processes can interact and communicate with tuple insertion alone. Tuple deletion is therefore not required for communication. However, only limited forms of communication can be achieved in this way. This is because for a given data domain, the database can only hold a polynomial number of tuples. The database therefore saturates after a polynomial number of inserts, so only a polynomial number of messages can be sent. This limitation is reflected in Theorems 5.3 and 5.4 below, which show that without both insertion and deletion, the data complexity of $\mathcal{TD}$ drops from RE to within PSPACE. In contrast, with both insertion and deletion, an unbounded number of messages can be sent, since tuples can be inserted and deleted from the database an unbounded number of times. In other words, once a message has been sent, it can be erased from the database and sent again at a later time. This increased capacity for communication is reflected in the RE-completeness of Theorem 5.2.

Theorem 5.3 *With tuple testing and tuple insertion as the only elementary operations, the data complexity of $\mathcal{TD}$ is in PSPACE.*

Theorem 5.4 *With tuple testing and tuple deletion as the only elementary operations, the data complexity of $\mathcal{TD}$ is in PSPACE.*

Observe that Theorem 5.3 corresponds to workflows in which information is added to a database but never deleted. This is a common situation in scientific workflows, where the workflow activities are laboratory experiments that gather, store and analyze information [8]. An example is the workflows at the Whitehead Institute/MIT Center for Genome Research [28, 79], where experimental results are accumulated in the database, and queried by analysis programs, but never deleted or altered. Note, however, that Examples 3.2, 3.3 and 6.4 all use deletion. Thus, even if application data is never deleted, deletion may still be needed for workflow modelling and simulation.

To close this section, we note that Theorem 5.2 shows that with three of the four kinds of elementary operation, $\mathcal{TD}$ can express *some* RE-complete transaction. Nevertheless, there are many transactions that it cannot express, even transactions with very low complexity. This is because without the fourth elementary operation (emptiness testing), $\mathcal{TD}$ can only express *monotonic* transactions. A monotonic transaction has the following property: if it terminates and commits when started from database $\mathbf{D}$, then it terminates and commits when started from any database containing $\mathbf{D}$. In contrast, emptiness testing is a *non-monotonic* transaction: $q.empty$ commits only when relation q is empty. Thus, even though $\mathcal{TD}$ with three elementary operations can simulate an arbitrary Turing machine, it *cannot* determine whether a base relation is empty, simply because this is a non-monotonic operation.[8] To express non-monotonic transactions, we must add a source of non-monotonicity to the language. Theorem 5.5 shows that emptiness testing is enough: adding it as a fourth

[8]This is a common phenomenon in logical languages. For instance, even though classical Horn logic (without negation) is data complete for RE, it cannot compute the difference of two base relations, simply because this is a non-monotonic query.

elementary operation enables $\mathcal{TD}$ to express *all* computable transactions, both monotonic and non-monotonic, as long as they are safe.[9]

Theorem 5.5 (Expressive Completeness)
With all four elementary operations, $\mathcal{TD}$ is expressively complete for safe transactions, i.e., it can express every safe transaction in RE.

5.2 Process-Oriented Features

This section explores the effect of concurrency and recursive processes on the data complexity of $\mathcal{TD}$. (Due to lack of space, we do not explore the effects of isolation.) We first show that without any restrictions on recursion, $\mathcal{TD}$ is very sensitive to concurrency, since a little concurrency can create a huge increase in complexity. We then develop a restricted form of recursion that is more robust. It allows iterated workflows to be specified, and concurrency has no effect on its complexity.

Concurrency is essential to the power of $\mathcal{TD}$. As Theorem 5.6 shows, when concurrent composition is removed, the data complexity of $\mathcal{TD}$ plummets from RE to EXPTIME. This version of the language, which we call *sequential* $\mathcal{TD}$, is comparable in complexity to many safe transaction languages [4, 5, 1, 32]. The main difference is that such languages are typically complete for PSPACE, not EXPTIME. The extra power of sequential $\mathcal{TD}$ comes from an ability to simulate *alternating* PSPACE machines [34].

Theorem 5.6 *Sequential $\mathcal{TD}$ is data complete for EXPTIME.*

Theorem 5.6 says that sequential $\mathcal{TD}$ expresses *some* transaction that is EXPTIME-complete. The next theorem strengthens this result.

Theorem 5.7 (Expressive Completeness) *Sequential $\mathcal{TD}$ can express every safe transaction in EXPTIME.*

Theorems 5.2 and 5.6 show that concurrency is essential to the RE-completeness of $\mathcal{TD}$. However, surprizingly little concurrency is required. For instance, recursion through concurrency is not needed. Nor is it necessary to spawn new processes at runtime. In fact, examining the proof of Theorem 5.2 shows that there need not be any concurrency in the rulebase at all! Concurrency is needed only in the *goal.* (Recall that a $\mathcal{TD}$ program consists of two parts: a rulebase, $\mathbf{P}$, and a goal, ϕ.) More specifically, it is enough for the goal to have the form $\phi_1 \mid \phi_2 \mid \phi_3$, where each ϕ_i is sequential. This means that RE-completeness can be achieved by *three* sequential processes executing concurrently. In the proof, these three processes are used to simulate a 2-stack machine [56], where two of the processes encode the stacks, and the third process encodes the finite control. We therefore have the following result.

Corollary 5.8 *$\mathcal{TD}$ programs with sequential rulebases are data complete for RE.*

[9] As usual in expressibility results of this kind, we assume that the constant symbols are uninterpreted. Formally, this means extending the notion of *genericity* from queries to transactions, as in [4, 5]. Details will soon be available in the appendices.

Restrictions on recursive processes also have a dramatic effect on data complexity. As Theorem 5.9 shows, if we eliminate recursion altogether, then data complexity plummets from RE to less than PTIME. Without recursion, we can still specify workflows like those in Examples 3.1 and 3.4, but we cannot simulate their execution on a set of work items, as in Example 3.2.

Theorem 5.9 *Non-recursive $\mathcal{TD}$ is in LOGSPACE.*

Together, Theorem 5.9 and Corollary 5.8 establish two extreme points, one of high complexity (RE), and one of low complexity (LOGSPACE). These two extremes correspond to unrestricted recursion and no recursion, respectively. Our last result in this section establishes an intermediate point (PSPACE). It allows more concurrency than Corollary 5.8, and more recursion than Theorem 5.9. It is based on a syntactic restriction we call *sequential tail recursion*. There are two essential elements to this restriction: one corresponds to tail recursion in classical logic programming, and the other forbids recursion through concurrent composition.

Definition 5.10 (Sequential Tail Recursion) Let $\mathbf{P}$ be a $\mathcal{TD}$ rulebase. A rule in $\mathbf{P}$ exhibits *sequential tail recursion* if it has the form $p \leftarrow \psi \otimes q$, where ψ is a goal, and q is the only atom in the rule body that is mutually recursive with p. The rulebase $\mathbf{P}$ exhibits sequential tail recursion if every recursive rule in $\mathbf{P}$ does. □

Observe that concurrent composition is allowed in the goal ψ in Definition 5.10. This goal may therefore contain concurrent processes that interact and communicate, like the workflows in Examples 3.1 and 3.4. Tail recursion allows these workflows to be iterated. That is, they can be executed over-and-over again until some condition is satisfied. For instance, a workflow for a shipping company may load goods onto a truck until the truck is full. Likewise, in a scientific laboratory, an experimental protocol may be repeated until a conclusive result is achieved. This is the case for the genome workflow described in [29]. Note that sequential tail recursion allows processes to be created and destroyed at runtime, but only inside ψ. In particular, the number of processes does not grow with each recursive call, as in the simulation of Example 3.2.

Example 5.11 (Iterated Workflows) Let $workflow(W)$ be the predicate defined in Example 3.1, which specifies a simple workflow. Suppose that each instance of the workflow records the result of its execution by inserting an atom of the form $result(W, R)$ into the database. Intuitively, this atoms means that R is the result of executing the workflow to work item W. For simplicity, we assume that R has two possible values: *conclusive* and *inconclusive*. The rules below execute the workflow on W over-and-over again until the result is conclusive. Observe that the rulebase (including the rules in Example 3.1) exhibits sequential tail recursion, even though $workflow(W)$ executes tasks concurrently.

$$
\begin{aligned}
start(W) &\leftarrow workflow(W) \otimes repeat(W) \\
repeat(W) &\leftarrow result(W, inconclusive) \otimes workflow(W) \otimes repeat(W) \\
repeat(W) &\leftarrow result(W, conclusive)
\end{aligned}
$$

The first rule executes the workflow. The second rule executes the workflow over and over again as long as the result of the previous execution was inconclusive. Finally, the third rule terminates the iteration when the result is conclusive. □

Theorem 5.12 *$\mathcal{TD}$ with sequential tail recursion is data complete for PSPACE.*

Finally, we show that the complexity of sequential tail recursion is insensitive to concurrency. In fact, the lower bound in Theorem 5.12 does not require concurrent composition at all.

Corollary 5.13 *Sequential $\mathcal{TD}$ with sequential tail recursion is data complete for PSPACE.*

6 Fully Bounded $\mathcal{TD}$

Section 5 established the effect of simple syntactic restrictions on the data complexity of workflows. Each of these restrictions targets a single syntactic feature of $\mathcal{TD}$, such as queries, updates, concurrency, or recursion. In practice, however, each of these features has an important role in the modeling of workflows and business processes, so we do not want to entirely eliminate any one of them. To address this concern, this section develops a more-complex restriction, called *full boundedness*, that has relatively-low complexity, but retains a wide range of modeling capabilities. We give only a brief and informal description of the restriction here, and state the main results. A formal development will soon be available in the appendices. An extended example based on cooperating workflows in a genome laboratory is given at the end of this section.

Full boundedness consists of two constraints: boundedness (which limits recursion), and acyclicity (which limits data flow). We discuss each constraint in turn.

Intuitively, a recursive predicate, p, is *bounded* by a base predicate, b, if the extent of b decreases with each recursive call to p. Thus, when executed in isolation, the number of recursive calls to p is bounded above by the extent of b, *i.e.*, by the number of tuples in b. For instance, in Example 3.2, predicate *simulate* is bounded by predicate *item*, since each recursive call to *simulate* deletes a tuple from *item*. Boundedness decreases the complexity of sequential rulebases from EXPTIME to NP, as the following theorem shows.

Theorem 6.1 *Bounded, sequential $\mathcal{TD}$ is data complete for NP.*

Unfortunately, by itself, boundedness has no effect on the complexity of concurrent rulebases. For instance, consider the following bounded rulebase:

$$\begin{aligned} p &\leftarrow item_1(W) \otimes del.item_1(W) \otimes task_1(W) \otimes ins.item_2(W) \otimes p \\ q &\leftarrow item_2(W) \otimes del.item_2(W) \otimes task_2(W) \otimes ins.item_1(W) \otimes q \end{aligned} \tag{1}$$

Observe that predicate p is bounded by $item_1$, and predicate q is bounded by $item_2$. Thus, when executed in isolation, p and q are each guaranteed to terminate. However, this guarantee disappears when p and q are executed *concurrently*. In this case, p moves tuples from $item_1$ to $item_2$, and q moves them back again. Thus, depending on how p and q are interleaved, predicates $item_1$ and $item_2$ may never become empty, so p and q may execute forever. Boundedness thus provides no guarantee of termination for concurrent programs. In fact, the following result shows that, by itself, boundedness has no effect on the complexity of $\mathcal{TD}$.

Corollary 6.2 *Bounded $\mathcal{TD}$ is data complete for RE.*

Thus, to reduce the complexity of concurrent programs, we need additional syntactic restrictions. The above example suggests a solution: if predicate p is bounded by b, then tuples removed from b should not eventually find their way back into b. We can easily guarantee this by defining a kind of data flow graph that keeps track of how tuples are moved about in the database. Termination is then guaranteed when the graph is acyclic. The data flow graph of rulebase (1) is cyclic. In contrast, the data flow graph of the following rulebase is acyclic:

$$p \leftarrow item_1(W) \otimes del.item_1(W) \otimes task_1(W) \otimes ins.item_2(W) \otimes p$$

$$q \leftarrow item_2(W) \otimes del.item_2(W) \otimes task_2(W) \otimes ins.item_3(W) \otimes q$$

Here, p is bounded by $item_1$, and q is bounded by $item_2$. In addition, the rulebase has a simple, linear data flow: p moves tuples from $item_1$ to $item_2$, and q moves tuples from $item_2$ to $item_3$. In cases like this, when the rulebase is bounded *and* it has an acyclic data flow graph, we say that it is *fully bounded.* The additional constraint of acyclicity reduces the complexity of bounded rulebases from RE to NP, as the following theorem shows.

Theorem 6.3 *Fully bounded $\mathcal{TD}$ is data complete for NP.*

Observe that full boundedness reduces the complexity of $\mathcal{TD}$ programs more than most other restriction considered in this paper. In fact, only two restrictions have lower complexity—eliminating recursion, and eliminating updates—and they severely reduce the ability of $\mathcal{TD}$ to model processes. In contrast, full boundedness retains a wide range of modeling capabilities. For instance, it is not restricted to tail recursion, and it allows recursion through sequential and concurrent composition. It also allows two-way communication between processes, concurrent access to shared resources, unlimited use of isolation, and all four elementary database operations. This means that a wide variety of practical workflows can be specified and simulated, including all of the examples in this paper. In addition, full boundedness allows separate workflows to be "hooked up" into a network of interacting workflows, as illustrated below.

6.1 Cooperating Workflow Networks

An enterprise may consist of not just a single workflow, but a collection of cooperating workflows. This is typically the case when one workflow prepares items needed by another workflow. The result is often a network of loosely-coupled workflows. For instance, in automobile manufacturing, one workflow may assemble carburetors, another transmissions, another exhaust systems, etc. The products of these individual workflows may then be fed to another workflow that assembles them into a finished automobile. Obviously, workflows may be cascaded and combined in this way to produce a complex network of workflows. Note that in this manufacturing example, the movement of items from one workflow to the next is acyclic. In such cases, the entire network of workflows can be represented by a fully bounded $\mathcal{TD}$ program.

This idea is illustrated in the next example, which involves three interacting workflows. Two of the workflows process items, which are then combined and further processed by the third workflow. The example is based on a common situation in large genome laboratories: some workflows prepare samples and reagents, while other workflows determine how they interact. A typical situation would be the following: one workflow processes long DNA samples, a second workflow processes short DNA samples, and a third workflow selects pairs of long and short samples that seem likely to overlap, and processes the two samples together to determine the exact nature of their overlap.

Example 6.4 (Cooperating Workflows) The rules below simulate three interacting workflows, represented by the predicates $workflow_1(I)$, $workflow_2(J)$ and $workflow_3(I, J)$. Here, $workflow_1$ and $workflow_2$ process items which are then combined and processed together by $workflow_3$. To transfer items between workflows, $workflow_1$ and $workflow_2$ place processed items in "baskets," from which $workflow_3$ then selects suitable *pairs* of items. A pair of items, I, J, is suitable if the predicate $suitable(I, J)$ is true. This predicate is an arbitrary database query, and may be defined by a complex set of $\mathcal{TD}$ rules.

The first set of rules below simulates the execution of $workflow_1$ and $workflow_2$, which we call the "producer" workflows. As in Example 3.2, we assume that information describing the work items is stored in the database, and can be accessed by the workflows as they execute. In particular, we assume that an identifier for each item to be processed by $workflow_i$ is initially stored in the base predicate $in_i(W)$, which acts as an "in basket" for the workflow. As the simulation progresses, work items are removed from the in basket, processed by the workflow, and then placed in an out basket, represented by the base relation $out_i(W)$.

$$
\begin{aligned}
produce_i &\leftarrow simulate_i \otimes ins.finished_i \\
simulate_i &\leftarrow getItem_i(W) \otimes [simulate_i \mid workflow_i(W) \otimes putItem_i(W)] \\
simulate_i &\leftarrow in_i.empty \\
getItem_i(W) &\leftarrow \odot[in_i(W) \otimes del.in_i(W)] \\
putItem_i(W) &\leftarrow ins.out_i(W)
\end{aligned}
$$

The first rule invokes the simulation of $workflow_i$, and inserts the atom $finished_i$ into the database when the simulation is finished. The second rule carries out the actual simulation. It recursively removes one work item after another from the in basket for $workflow_i$. After each removal, the rule spawns a workflow instance to process the item, where different workflow instances execute concurrently. After the workflow instance terminates, the work item is put into the out basket. The third rule terminates the recursion when there are no work items left in the in basket.

The next set of rules simulates the execution of $workflow_3$, which we call the "consumer" workflow. The out baskets for $workflow_1$ and $workflow_2$ act as in baskets for $workflow_3$. This workflow selects a pair of items, one from each basket, and processes them together. This process is repeated until one of the baskets is permanently empty, at which point the workflow stops.

$$consume \leftarrow simulate_3 \otimes ins.finished_3$$

$$simulate_3 \leftarrow selectItems(I,J) \otimes [simulate_3 \mid workflow_3(I,J)]$$

$$simulate_3 \leftarrow finished_1 \otimes basket_1.empty$$

$$simulate_3 \leftarrow finished_2 \otimes basket_2.empty$$

$$selectItems(I,J) \leftarrow \odot[out_1(I) \otimes out_2(J) \otimes suitable(I,J) \otimes del.out_1(I) \otimes del.out_2(J)]$$

The first rule invokes the simulation of $workflow_3$, and inserts the atom $finished_3$ into the database when the simulation is finished. The second rule carries out the actual simulation. It first selects and removes a pair of items, I and J, from baskets out_1 and out_2, respectively. After each selection, the rule spawns a workflow instance, $workflow_3(I,J)$, to process the pair. The third and fourth rules terminate the recursion when one of the baskets is permanently empty. Specifically, the third rule stops the simulation when $workflow_1$ has finished and basket out_1 is empty. Likewise, the fourth rule stops the simulation when $workflow_2$ has finished and basket out_2 is empty. The fifth rule does the actual work of selecting and removing a pair of items from the two baskets.

Finally, the following rule executes all three workflows concurrently:

$$work \leftarrow produce_1 \mid produce_2 \mid consume$$

□

7 Related Work

$\mathcal{TD}$ can be compared to other languages in a number of ways; *e.g.*, in terms of workflow, complexity, or semantics. A comprehensive comparison of the semantics of $\mathcal{TD}$ with other update languages and logics of action can be found in [24, 22]. Here, we focus on comparisons based on workflow and complexity, the two themes of this paper. Due to space limitations, we confine the discussion to work in database theory. Related work in other areas was discussed briefly in Section 1. Additional discussion can be found in [19, 21].

Workflow: The database-theory community has recently begun to study business processes and workflow. For instance, Wodtke and Weikum have used state charts [49] to develop a formal foundation for workflow execution in a distributed environment [82]. The goal of this research is to partition a workflow (at compile time) into "orthogonal" components that can be executed on different servers in a distributed environment. To simplify the problem, nested states (*i.e.*, sub-workflows) are not considered. This work is clearly orthogonal to our own.

Abiteboul, Vianu et al have developed relational transducers for use in electronic commerce. Like the transducers in language theory, a relational transducer can be viewed as a single sequential process. The main novelty is that a state in a relational transducer is a relational database, and the state transition function is a Datalog-like program. A relational transducer accepts a sequence of tuple sets as input, and produces a sequence of tuple sets as output. The work itself is primarily concerned with *individual* transducers and their properties, such as log validation and goal reachability. The work does not address many of the central issues in process modeling, such as the *interaction* between concurrently executing processes (or transducers), and the *decomposition* of complex processes into simpler subprocesses.

Davulcu, Kifer et al have used Transaction Datalog ($\mathcal{TD}$) and the larger framework of Concurrent Transaction Logic ($\mathcal{CTR}$) to specify and reason about workflows [38]. This research was the first attempt to apply $\mathcal{TD}$ and $\mathcal{CTR}$ to workflow. It focuses on compiling global constraints (specified in $\mathcal{CTR}$) into workflow graphs (specified in $\mathcal{TD}$), and establishes results on consistency and verification. To make the problem tractable, strong restrictions were placed on the kind of $\mathcal{TD}$ programs allowed. Our paper is orthogonal to this work. We place no restrictions on the allowed $\mathcal{TD}$ programs, but we do not consider global constraints. We also address a different set of questions. In particular, we ask what kind of workflows can be expressed in $\mathcal{TD}$, and how is this affected by the interaction of data-oriented and process-oriented features.

Complexity and Expressiveness: Using the results of Sections 5 and 6, we can compare $\mathcal{TD}$ to other query and update languages based on their data complexity and expressive power. For instance, we can compare $\mathcal{TD}$ to the highly expressive languages described in [2]. Like $\mathcal{TD}$, these languages are data complete for RE, although in each case, the source of power is different. For instance, the language $while_N$ can perform complex arithmetic operations, the language $while_{new}$ can create new constant symbols during execution, and the language $while_{uty}$ can repeatedly increase the arity of database relations. The version of $\mathcal{TD}$ presented here has none of these abilities. Instead, its power comes from *cooperative concurrency*, *i.e.*, the ability to execute programs concurrently and have them communicate, synchronize, or otherwise interact [50]. This feature is essential to workflow, but is lacking in traditional database languages, including all the languages described in [2].

If we remove concurrency from $\mathcal{TD}$, then its data complexity drops from RE to EXPTIME (Theorem 5.6). The resulting language (sequential $\mathcal{TD}$) can be compared to the less expressive languages in [2]. For instance, sequential $\mathcal{TD}$ has many features in common with the basic *while* language, such as insertion, deletion and sequential composition. However, whereas the *while* language is data complete for PSPACE, sequential $\mathcal{TD}$ is data complete for EXPTIME. This difference in complexity is due to a basic programming feature—recursive subroutines—that $\mathcal{TD}$ supports, but the *while* language does not. Instead, the *while* language supports iteration, which corresponds to a special form of recursion (tail recursion), and allows the *while* language to simulate PSPACE machines. By using more general forms of recursion, sequential $\mathcal{TD}$ can simulate *alternating* PSPACE machines. This accounts for its greater computational power, since alternating PSPACE = EXPTIME [34].

In addition to data complexity, Section 5 establishes results on expressive completeness (Theorems 5.5 and 5.7). Unlike some expressiveness results in the literature (*e.g.*, [61, 81, 33, 3, 2]), these results do *not* assume the data domain is linearly ordered. The assumption of ordered domains is a technical device that is often used to achieve expressiveness results, but it is not an intrinsic feature of databases [3]. Because our results do not rely of this assumption, they provide a complete characterization of the safe transactions in two well-known complexity classes (RE and EXPTIME). Note that these results are about standard complexity classes, not the *relational* complexity classes introduced in [3], which are based on so-called relational Turing machines. Relational Turing machines capture many of the important features of database programming, but they are strictly weaker than ordinary Turing machines. For instance, relational Turing machines (no matter how powerful) cannot determine whether a database has an even or odd number of tuples [3]. In contrast, this query *can* be expressed by $\mathcal{TD}$, as well as by sequential $\mathcal{TD}$ and fully bounded $\mathcal{TD}$, as shown in the appendices. In fact, because sequential $\mathcal{TD}$ expresses all the safe transactions in EXPTIME (Theorem 5.7), it is strictly more expressive than all the fixpoint logics described in [3], including those based on alternating fixpoints. This is because their expressiveness is bounded above by the relational version of EXPTIME, which is strictly smaller than EXPTIME proper [3].

Finally, it is worth noting that our notion of transaction expressiveness is different from that developed by Abiteboul and Vianu in [4, 5]. We have defined expressiveness in terms of the ability to *recognize* whether a given database pair $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$ belongs to a transaction. In contrast, Abiteboul and Vianu define expressiveness in terms of the ability to *generate* $\mathbf{D}_2$ from $\mathbf{D}_1$. As a practical matter, one can use the operational semantics of $\mathcal{TD}$ to generate $\mathbf{D}_2$ from $\mathbf{D}_1$, as in our implementation. However, as a theoretical device, we have found that defining expressiveness in terms of recognition simplifies the formal development, and leads to clean results on the expressiveness and complexity of $\mathcal{TD}$. Moreover, these results are all formulated in terms of standard complexity classes, such as NP, PSPACE and RE. In contrast, Abiteboul and Vianu introduce special classes such as DB-PSPACE and NDB-PSPACE. These classes are defined in terms of standard Turing machines, but in a non-standard way. The idea is to view a Turing machine not as recognizing a language, but as computing a mapping from input to output.[10] For instance, NDB-PTIME is the set of (non-deterministic) mappings computable by NP machines. Note that transactions in NDB-PTIME are easy to compute: they execute in polynomial time, making non-deterministic choices along the way. In contrast, a transaction whose recognition problem is in NP can be very hard to compute. This is certainly the case for $\mathcal{TD}$ programs whose data complexity is NP-complete. As another example, the transactions recognizable in PSPACE are the same as those recognizable in NPSPACE, since PSPACE = NPSPACE [56]. In contrast, DB-PSPACE $\neq$ NDB-PSPACE, since DB-PSPACE contains only deterministic transactions, while NDB-PSPACE contains non-deterministic transactions. These differences reflect the two different notions of expressiveness: one based on output recognition, and the other based on output generation.

Acknowledgments: Thanks go to David Toman and Michael Kifer for comments on an earlier version of this work. This work was supported in part by a Research Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC), and by the Department of Computer and Information Science at the University of Pennsylvania.

[10] Curiously, although this idea leads to a notion of expressive completeness, there seems to be no notion of data complexity. At least, such a notion was not developed in [4, 5].

References

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

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Abiteboul, M.Y. Vardi, and V. Vianu. Fixpoint logics, relational machines, and computational complexity. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pages 156–168, Boston, MA, June 22–25 1992. IEEE Computer Society Press.

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

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

[6] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *ACM Symposium on Principles of Database Systems*, pages 179–187, Seattle, Washington, June 1998.

[7] N. Adam, V. Atluri, and W. Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, March 1998.

[8] *Standard Guide for Laboratory Information Management Systems (LIMS)*. American Society for Testing and Materials, 1916 Race St., Philadelphia PA 19103, U.S.A, 1993.

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

[10] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[11] T. Baker, J. Gill, and R. Soloway. Relativizations of the $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ question. *SIAM Journal of Computing*, 4:431–442, 1975.

[12] J. Balcazar, J. Gabarro, and M. Santha. Deciding bisimilarity is P-complete. *Formal Aspects of Computing*, 4(6):638–648, 1992.

[13] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

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

[15] A.J. Bonner. Concurrent transaction logic. Presented at the *Dagstuhl Seminar on Transactional Workflows*, July 15–19 1996, International Conference and Research Center for Computer Science, Schloss Dagstuhl, Wadern, Germany.

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

[17] A.J. Bonner. Intuitionistic deductive databases and the polynomial time hierarchy. *Journal of Logic Programming*, 33(1):1–47, October 1997.

[18] A.J. Bonner. A logical semantics for hypothetical rulebases with deletion. *Journal of Logic Programming*, 32(2):119–170, August 1997.

[19] A.J. Bonner. Transaction Datalog: a compositional language for transaction programming. In *Proceedings of the International Workshop on Database Programming Languages*, number 1369 in LNCS, pages 373–395. Springer-Verlag, 1998. Workshop held in Estes Park, Colorado, August 1997. Long version available at `http://www.cs.toronto.edu/~bonner/papers.html#transaction-logic`.

[20] 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`.

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

[22] 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.

[23] A.J. Bonner and M. Kifer. Results on reasoning about updates in transaction logic. In [45]. 1998.

[24] A.J. Bonner and M. Kifer. The state of change: A survey. In [45]. 1998.

[25] 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.

[26] A.J. Bonner, L.T. McCarty, and K. Vadaparty. Expressing Database Queries with Intuitionistic Logic. In *North American Conference on Logic Programming*, pages 831–850, Cleveland, Ohio, October 16–20 1989. MIT Press.

[27] A.J. Bonner, A. Shrufi, and S. Rozen. Benchmarking object-oriented DBMSs for workflow management. In *OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX, October 15 1995.

[28] A.J. Bonner, A. Shrufi, and S. Rozen. Database requirements for workflow management in a high-throughput genome laboratory. In *NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, pages 119–125, Athens, GA, May 8–10 1996.

[29] A.J. Bonner, A. Shrufi, and S. Rozen. LabFlow-1: a database benchmark for high-throughput workflow management. In *Int'l Conference on Extending Database Technology*, number 1057 in LNCS, pages 463–478, Avignon, France, March 25–29 1996. Springer-Verlag. Full technical report available at `http://www-genome.wi.mit.edu/informatics/informatics_papers/bibliography.html`.

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

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

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

[33] A.K. Chandra and D. Harel. Horn Clause Queries and Generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[34] A.K. Chandra, D. Kozen, and L.J. Stockmeyer. Alternation. *Journal of ACM*, 28:114–133, 1981.

[35] S. Christensen, Y. Hirshfeld, and F. Moller. Decidable subsets of CCS. *The Computer Journal*, 37(4):233–242, 1994. Special issue on process algebra.

[36] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 244–263, 1986.

[37] *Communications of ACM*, 34(11), November 1991. Special issue on the Human Genome Project.

[38] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, pages 25–33, Seattle, Washington, June 1998.

[39] U. Dayal and Q. Chen. From database programming to business process programming. In *Proceedings of the International Workshop on Database Programming Languages*, Gubbio, Umbria, Italy, September 1995. Springer-Verlag.

[40] 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.

[41] J. Desel. *Free Choice Petri Nets*. Cambridge University Press, 1995.

[42] D. Drusinsky and D. Harel. On the power of bounded concurrency I: Finite automata. *Journal of ACM*, 41(3):517–539, 1994.

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

[44] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 997–1072. Elsevier, 1990.

[45] B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors. *Transactions and Change in Logic Databases.* Number 1472 in LNCS. Springer-Verlag, Berlin, 1998. In press.

[46] 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.

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

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

[49] D. Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

[50] D. Harel. A thesis for bounded concurrency. In *Proceedings of the 14th Symposium on Mathematical Foundations of Computer Science*, number 379 in LNCS, pages 35–48. Springer-Verlag, 1989.

[51] D. Harel, O. Kupferman, and M.Y. Vardi. On the complexity of verifying concurrent transition systems. Technical Report CS97-01, Faculty of Mathematical Sciences, The Weizmann Institute of Science, Rehovot, Israel, 1997.

[52] D. Harel, R. Rosner, and M. Vardi. On the power of bounded concurrency III: Reasoning about programs. In *Int'l Symposium on Logic in Computer Science*, pages 478–488. IEEE Press, New York, 1990.

[53] M. Hennessy. *An Algebraic Theory of Processes.* MIT Press, Cambridge, MA, 1988.

[54] T. Hirst and D. Harel. On the power of bounded concurrency II: Pushdown automata. *Journal of ACM*, 41(3):540–554, 1994.

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

[56] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

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

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

[59] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *Proceedings of the ACM International Joint Conference on Work Activities Coordination and Collaboration (WACC)*, pages 69–78, San Francisco, February 22–25 1999.

[60] Samuel Y.K. Hung. Implementation and Performance of Transaction Logic in Prolog. Master's thesis, Department of Computer Science, University of Toronto, 1996.

[61] N. Immerman. Relational Queries Computable in Polynomial Time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 147–152, 1982.

[62] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.

[63] K. Jensen. *Colored Petri Nets*. Springer-Verlag, Berlin, 1996.

[64] K. Jensen and G. Rozenberg, editors. *High Level Petri Nets*. Springer-Verlag, Berlin, 1991.

[65] N.D. Jones, L.H. Landweber, and Y.E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4:277–299, 1977.

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

[67] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

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

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

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

[71] W.F. Ogden, W.E. Riddle, and W.C. Rounds. Complexity of expressions allowing concurrency. In *ACM Symposium on Principles of Programming Languages*, pages 185–194, 1978.

[72] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[73] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, Berlin, 1985.

[74] 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.

[75] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD Conference on Management of Data*, pages 442–453, New York, May 1994. ACM.

[76] E. Shapiro, editor. *Concurrent Prolog*, volume 1. MIT press, Cambridge, MA, 1987.

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

[78] Amalia Sleghel. Implementation of Concurrent Transaction Logic. Master's thesis, Department of Computer Science, University of Toronto. Forthcoming.

[79] 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.

[80] D. Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*. Number 369 in LNCS. Springer-Verlag, 1989.

[81] M.Y. Vardi. The complexity of relational query languages. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 137–146, May 1982.

[82] Dirk Wodtke and Gerhard Weikum. A formal foundation for distributed workflow execution based on state charts. In *Int'l Conference on Database Theory*, pages 230–246, 1997.

[83] D. Worah and A. Sheth. Transactions in transactional workflows. In [62], chapter 1, pages 3–45. Kluwer Academic Publishers, 1997.

A APPENDIX: Logical Semantics

Recall from Section 2 that Transaction Datalog ($\mathcal{TD}$) is a fragment of Concurrent Transaction Logic ($\mathcal{CTR}$). $\mathcal{TD}$ therefore inherits the logical semantics of $\mathcal{CTR}$. This semantics provides a declarative interpretation of $\mathcal{TD}$ programs, which complements the procedural interpretation illustrated in Section 3. The declarative and procedural interpretations are linked by the proof theory of $\mathcal{CTR}$, which executes $\mathcal{TD}$ programs by proving theorems. The logical semantics and proof theory of $\mathcal{CTR}$ have been developed in detail elsewhere [21, 19]. This appendix and the next review the basic ideas, in a slightly simplified form.

A.1 Overview of Concurrent Transaction Logic

Concurrent Transaction Logic is an extension of first-order classical logic for reasoning about database transactions and business processes. It extends classical logic with three new operators: *serial conjunction*, denoted $\otimes$; *concurrent conjunction*, denoted $|$; and a *modality of isolation*, denoted $\odot$. Syntactically, if α and β are logical formulas in $\mathcal{CTR}$, then so are $\alpha \otimes \beta$, $\alpha \mid \beta$ and $\odot\alpha$. Logical formulas may also be constructed from the connectives and quantifiers of classical logic. Thus $\neg\alpha$, $\alpha \vee \beta$ and $\forall X\, \alpha$ are logical formulas in $\mathcal{CTR}$.

To illustrate the semantics of these formulas, suppose that $withdraw(N, A)$ is an atomic formula that intuitively means, "*N dollars was withdrawn from bank account A.*" Here are two formulas involving *withdraw*:

$$withdraw(50, 123) \vee withdraw(50, 456) \qquad\qquad \exists A\ withdraw(50, A)$$

These formulas use classical connectives only, and their meaning in Concurrent Transaction Logic is the same as in classical logic. The first formula says, "*$50 was withdrawn from account 123 or from account 456.*" The second formula says, "*$50 was withdrawn from some account.*" Here are examples with non-classical connectives:

$$withdraw(50, 123) \otimes withdraw(75, 456) \qquad\qquad withdraw(50, 123) \mid withdraw(75, 456)$$

These formulas make statements about how and when actions take place. The first formula says, "*$50 was withdrawn from account 123, and then $75 was withdrawn from account 456.*" The second formula says, "*$50 was withdrawn from account 123, and concurrently, $75 was withdrawn from account 456.*" Finally, consider the formula $\odot\, withdraw(50, 123)$. It says, "*$50 was withdrawn from account 123 in isolation.*" Isolation is a fundamental property of database transactions [14, 48]. Intuitively, an action is *isolated* if it does not interact or communicate with other actions. In this case, it means that while $50 was withdrawn from account 123, no other action accessed the account.

Simple formulas like those above can be combined into more-complex formulas. For instance,

$$\neg\, [withdraw(50, 123) \mid withdraw(50, 456)] \qquad \forall N, A\ [withdraw(N, A) \rightarrow \odot\, withdraw(N, A)]$$

The first formulas says, "*$50 was not concurrently withdrawn from accounts 123 and 456.*" The second formulas says, "*Whenever money was withdrawn from an account, the withdrawal took place in isolation.*" With such formulas, we can express properties of transaction programs and reason about them [23].

A.2 Execution Traces

The semantics of $\mathcal{CTR}$ describes the effect of a program on a database. For isolated transactions, this effect can be described simply as a mapping from databases to databases, or more generally, as a binary relation on databases, as in [4, 5, 1]. However, for programs that interact and communicate, this is not enough, since the effect of a program can depend on other programs that execute concurrently with it.

In an interleaving model of concurrency, only one program executes at a time, while all concurrent programs are suspended. To represent such behavior, we record the state of the database whenever a program is suspended or awakened. Formally, we represent the effect of a program, ϕ, as a list of database 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 a *trace*. In this list, 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. For example, the list $[\{a\}\{ab\}, \{d\}\{cd\}]$ is an execution trace of the program $ins.b \otimes ins.c$. Starting from state $\{a\}$, the program first inserts b, changing the database to state $\{ab\}$. The program is then suspended, and other programs change the database to state $\{d\}$. The program is then re-awakened, and it inserts the atom c, leaving the database in state $\{cd\}$. If a program executes in isolation, then it need not be interleaved with any other program: it can execute continuously, without interruption or suspension. The effect of an isolated program can thus be represented by a singleton trace, *i.e.*, an execution trace consisting of a single database pair, $\mathbf{D}_1\mathbf{D}_2$. For example, the list $[\{a\}\{abc\}]$ is an execution trace of the program $\odot(ins.b \otimes ins.c)$. Starting from state $\{a\}$, the program inserts the atoms b and c, leaving the database in state $\{abc\}$.

Note that execution traces represent only the *effects* of database programs, and not their actual execution inside a DBMS. For instance, execution traces do not record operations that are later aborted and undone, since these have no permanent effect on the database. Moreover, although we describe operations as being interleaved, a DBMS may actually execute them simultaneously (*i.e.*, in parallel). Such details are addressed by the procedural semantics of $\mathcal{TD}$ and by our implementation, but are purposely ignored in the declarative semantics, which is inherently high-level. (Likewise, the declarative semantics of classical Datalog specifies only the results of a query, and ignores the details of query processing and optimization.)

Using execution traces, we can develop a simple semantics for the three non-classical connectives of Concurrent Transaction Logic, $\otimes$, $\mid$ 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, if $\overline{\mathbf{D}}_1$ is a trace of ϕ_1, and $\overline{\mathbf{D}}_2$ is a trace of ϕ_2, then their concatenation is a trace of $\phi_1 \otimes \phi_2$, and any interleaving is a trace of $\phi_1 \mid \phi_2$.

Definition A.1 (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 all $1 \le i \le n$. In this case, $[\mathbf{D}_1\mathbf{D}_n']$ is the *reduction* of the trace. □

For example, the trace $[\mathbf{D}_1\mathbf{D}_2, \mathbf{D}_2\mathbf{D}_3, \mathbf{D}_3\mathbf{D}_4]$ is reducible, and its reduction is $[\mathbf{D}_1\mathbf{D}_4]$. Intuitively, if a program has a reducible trace, then the database does not change when the program is suspended. The suspensions are therefore unnecessary, and the program can execute continuously, without interruption. The reduced trace therefore represents another possible execution of the program—an isolated execution. For example, if $[\mathbf{D}_1\mathbf{D}_2, \mathbf{D}_2\mathbf{D}_3, \mathbf{D}_3\mathbf{D}_4]$ is an execution trace of ϕ, then $[\mathbf{D}_1\mathbf{D}_4]$ is an execution trace of $\odot\phi$.

A.3 Model Theory

Logical formulas in $\mathcal{CTR}$ are evaluated on execution traces. To formalize this idea, we define the basic semantic structures of $\mathcal{CTR}$ to be *mappings* from execution traces to sets of atoms (Definition A.2). Such a mapping specifies what atoms are true on what traces. Building on this, we then define what formulas are true on what traces (Definition A.3). Finally, we develop the standard logical notions of model and entailment for $\mathcal{CTR}$. In addition, we develop a new notion called *executional entailment*, which provides the logical foundation for executing $\mathcal{TD}$ programs, *i.e.*, logic programs with updates. Recall that a database state, $\mathbf{D}$, is a set of base facts. We define $dom(\mathbf{D})$ to be the set of constant symbols appearing in $\mathbf{D}$, which we call the *data domain* of $\mathbf{D}$.

Definition A.2 (Structures) A *structure* (or *$\mathcal{CTR}$ structure*) is a mapping $\mathbf{M}$ that assigns a set of atoms to every execution trace. This mapping is subject to the following restrictions, for every base predicate symbol, p, and all database states $\mathbf{D}, \mathbf{D}_1, \mathbf{D}_2$ such that $dom(\mathbf{D}_2) \subseteq dom(\mathbf{D}_1)$:

1. $p(\overline{x}) \in \mathbf{M}([\mathbf{D}\mathbf{D}])$ *if* $p(\overline{x}) \in \mathbf{D}$
2. $p.empty \in \mathbf{M}([\mathbf{D}\mathbf{D}])$ *if* $\mathbf{D}$ contains no atoms of the form $p(\overline{x})$
3. $ins.p(\overline{x}) \in \mathbf{M}([\mathbf{D}_1\mathbf{D}_2])$ *if* $\mathbf{D}_2 = \mathbf{D}_1 \cup \{p(\overline{x})\}$
4. $del.p(\overline{x}) \in \mathbf{M}([\mathbf{D}_1\mathbf{D}_2])$ *if* $\mathbf{D}_2 = \mathbf{D}_1 - \{p(\overline{x})\}$

□

Observe that each restriction in this definition corresponds to an elementary database operation.[11] In addition, the condition $dom(\mathbf{D}_2) \subseteq dom(\mathbf{D}_1)$ ensures that elementary updates do not insert new constant symbols into the database. The next definition defines what it means for a sentence (*i.e.*, a formula without free variables) to be true on a trace in a structure.

Definition A.3 (Satisfaction) Let $\mathbf{M}$ be a $\mathcal{CTR}$ structure, and π be an execution trace. Then,

1. $\mathbf{M}, \pi \models q$ if and only if $q \in \mathbf{M}(\pi)$, for any ground atomic formula, q.
2. $\mathbf{M}, \pi \models \neg\phi$ if and only if it is not the case that $\mathbf{M}, \pi \models \phi$.

[11] By changing the restrictions, we can easily change the elementary operations [21].

3. $\mathbf{M}, \pi \models \phi \wedge \psi$ if and only if $\mathbf{M}, \pi \models \phi$ and $\mathbf{M}, \pi \models \psi$.
4. $\mathbf{M}, \pi \models (\forall X)\phi$ if and only if $\mathbf{M}, \pi \models \phi\{X/t\}$ for *every* ground term, t.
5. $\mathbf{M}, \pi \models \phi \otimes \psi$ if and only if $\mathbf{M}, \pi_1 \models \phi$ and $\mathbf{M}, \pi_2 \models \psi$ for *some* traces π_1, π_2 whose concatenation is π.
6. $\mathbf{M}, \pi \models \phi \mid \psi$ if and only if $\mathbf{M}, \pi_1 \models \phi$ and $\mathbf{M}, \pi_2 \models \psi$ for *some* traces π_1, π_2 having π as an interleaving.
7. $\mathbf{M}, \pi \models \odot\phi$ if and only if $\mathbf{M}, \pi' \models \phi$ for some trace π' that reduces to π.

If $\mathbf{M}, \pi \models \phi$ then we say that ϕ is *satisfied* (or is *true*) on trace π in structure $\mathbf{M}$. $\square$

Definition A.4 (Models) A $\mathcal{CTR}$ structure, $\mathbf{M}$, is a *model* of a logical formula ϕ if $\mathbf{M}, \pi \models \phi$ for every trace π. In this case, we write $\mathbf{M} \models \phi$. A structure is a model of a set of formulas if it is a model of every formula in the set. $\square$

Definition A.5 (Logical Entailment) Let ϕ and ψ be two logical formulas of $\mathcal{CTR}$. Then ϕ *entails* ψ if every model of ϕ is also a model of ψ. In this case, we write $\phi \models \psi$. $\square$

We use standard abbreviations: $\phi \vee \psi$ means $\neg(\neg\phi \wedge \neg\psi)$; $\exists X\, \phi$ means $\neg \forall X \neg\phi$; and $\phi \leftarrow \psi$ means $\phi \vee \neg\psi$. Given these, here are two examples of logical entailment in $\mathcal{CTR}$:

$$\alpha \leftarrow \beta \;\models\; (\alpha \mid \gamma) \leftarrow (\beta \mid \gamma) \qquad\qquad (\alpha \vee \beta) \otimes \gamma \;\models\; (\alpha \otimes \gamma) \vee (\beta \otimes \gamma)$$

In [23], we show how to use logical entailment to reason about the properties of $\mathcal{TD}$ programs.

Finally, we introduce the notion of *executional entailment*, which defines the effect of an isolated program on a database. Such a program does not interact with other database programs, but it may have a rich structure of nested transactions and concurrent processes that interact and communicate among themselves. Executional entailment is the subject of the complexity results in Section 5. It also provides the logical foundation for a procedure that executes $\mathcal{TD}$ programs by proving theorems, as described in Appendix B.

Definition A.6 (Executional Entailment) Let ϕ be a logical formula, $\mathbf{P}$ be a set of logical formulas, and $\mathbf{D}_1$, $\mathbf{D}_2$ be database states. Then $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$ if $\mathbf{M}, [\mathbf{D}_1\mathbf{D}_2] \models \odot\phi$ for every model $\mathbf{M}$ of $\mathbf{P}$. $\square$

In Transaction Datalog, this definition can be interpreted procedurally, in terms of program execution. From this perspective, ϕ is a program invocation, and $\mathbf{P}$ is a set of program and subroutine definitions. The statement $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$ then says that an isolated execution of ϕ can change the database from state $\mathbf{D}_1$ to state $\mathbf{D}_2$. If at least one such $\mathbf{D}_2$ exists, then we say that the program *succeeds* from $\mathbf{D}_1$; otherwise, we say it *fails*. Here are three simple examples of executional entailment, valid for any rulebase, $\mathbf{P}$:

$$\mathbf{P}, \{a\}\{ab\} \models \mathit{ins.b} \qquad \mathbf{P}, \{a\}\{abc\} \models \mathit{ins.b} \otimes \mathit{ins.c} \qquad \mathbf{P}, \{a\}\{bc\} \models \mathit{ins.b} \otimes \mathit{ins.c} \otimes \mathit{del.a}$$

As another example, consider the concurrent goal $(\mathit{del.b} \otimes \mathit{del.c}) \mid (\mathit{ins.b} \otimes \mathit{ins.c})$. It is equivalent to an interleaving of the two sequential subgoals $\mathit{del.b} \otimes \mathit{del.c}$ and $\mathit{ins.b} \otimes \mathit{ins.c}$. There are six such interleavings. Formally, $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models (\mathit{del.b} \otimes \mathit{del.c}) \mid (\mathit{ins.b} \otimes \mathit{ins.c})$ is true if and only if at least one of the following is true:

$$\mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models del.b \otimes del.c \otimes ins.b \otimes ins.c \qquad \mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models ins.b \otimes ins.c \otimes del.b \otimes del.c$$
$$\mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models del.b \otimes ins.b \otimes del.c \otimes ins.c \qquad \mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models ins.b \otimes del.b \otimes ins.c \otimes del.c$$
$$\mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models del.b \otimes ins.b \otimes ins.c \otimes del.c \qquad \mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models ins.b \otimes del.b \otimes del.c \otimes ins.c$$

Thus, if $\mathbf{D}_1$ is the empty database, then $\mathbf{D}_2$ can be any of the databases $\{\}$, $\{b\}$, $\{c\}$ or $\{bc\}$. In addition, suppose that the rulebase, $\mathbf{P}$, contains the two rules $p \leftarrow del.b \otimes del.c$ and $p \leftarrow ins.b \otimes ins.c$. Then $\mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models p \mid q$ if $\mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models (del.b \otimes del.c) \mid (ins.b \otimes ins.c)$. Hence,

$$\mathbf{P},\{\}\{\} \models p \mid q \qquad \mathbf{P},\{\}\{b\} \models p \mid q \qquad \mathbf{P},\{\}\{c\} \models p \mid q \qquad \mathbf{P},\{\}\{bc\} \models p \mid q$$

B APPENDIX: Execution as Deduction

The declarative semantics developed in Appendix A determines whether a given database transformation is possible. That is, given two databases, $\mathbf{D}_1$ and $\mathbf{D}_2$, it determines whether a $\mathcal{TD}$ program can transform $\mathbf{D}_1$ into $\mathbf{D}_2$. This is what the expression $\mathbf{P},\mathbf{D}_1\mathbf{D}_2 \models \psi$ means. However, although it specifies what a $\mathcal{TD}$ program should do, the declarative semantics provides no way of actually *executing* $\mathcal{TD}$ programs, that is, of computing $\mathbf{D}_2$ from $\mathbf{D}_1$. To address this issue, this appendix develops a logical inference system. Given an initial state, $\mathbf{D}_1$, the system infers whether a program can succeed from $\mathbf{D}_1$, and if so, it computes the final state, $\mathbf{D}_2$. Moreover, it does this in a way that corresponds to the intuitive notion of execution in logic programming. The procedural semantics outlined in Appendix C is based on this inference system, as is our implementation [60, 78]. The details of the inference system have been published elsewhere [21]. This section reviews the main ideas.

As illustrated in Section 3 and Appendix C, a $\mathcal{TD}$ program can contain numerous sequential processes, each of which execute from left to right. To formalize this idea, we introduce the notion of *hot components*. These are the left-most formulas in a set of sequential processes, *i.e.*, the formulas that are ready to execute. For instance, in the serial goal $b \otimes c$, the atom b is ready to execute, but the atom c is not (since it must wait for b to finish). In the concurrent goal $b \mid c \mid d$, all three atoms, b, c, d, are ready to execute. Definition B.1 formalizes this idea, and Figure 1 gives additional examples.

Definition B.1 (Hot Components) If ψ is a goal, then its set of *hot components*, written $hot(\psi)$, is defined recursively as follows:

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

$\square$

A basic operation of the inference system is removing a hot component from a goal, or replacing it by some other formula. For instance, removing c from $(a \otimes b) \mid (c \otimes d)$ gives $(a \otimes b) \mid d$, and removing a from $(a \mid b) \otimes (c \mid d)$ gives $b \otimes (c \mid d)$. As shown in the last line

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

Figure 1: Some formulas and their hot components.

of Figure 1, a goal may have more than one occurrence of a given hot component. In this case, we remove or replace just a single occurrence. For instance, replacing an occurrence of a with ψ in $(a \otimes b) \mid (a \otimes c)$ can give $(\psi \otimes b) \mid (a \otimes c)$ and it can give $(a \otimes b) \mid (\psi \otimes c)$. When used in top-down mode (see below), the inference system repeatedly transforms the goal by removing or replacing hot components, until the empty goal is reached. For instance, by removing the atoms a, b, c in that order, the goal $(a \otimes b \otimes c)$ is transformed first to $(b \otimes c)$, then to (c), and finally to $(\,)$, which denotes the empty goal.

Formally, the inference system manipulates expressions of the form $\mathbf{P}, \mathbf{D} \text{---} \vdash \psi$, called *sequents*. Here, $\mathbf{P}$ is a rulebase, $\mathbf{D}$ is a database, and ψ is a goal. The informal meaning of such a sequent is that transaction ψ can *succeed* from $\mathbf{D}$, that is, $\mathbf{P}, \mathbf{D}\mathbf{D}' \models \psi$ for some database $\mathbf{D}'$. The inference system itself is a collection of axioms and inference rules. An axiom is a sequent, and an inference rule is a pair of sequents separated by a horizontal line. An inference rule has a simple interpretation: if the upper sequent can be inferred, then the lower sequent can also be inferred. Based on the axiom sequents, the system uses the inference rules to derive more-and-more sequents. In its most developed form [21], the inference system exploits unification and most general unifiers, in the logic programming tradition. However, to simplify the presentation, this section describes a simplified version of the system in which all goals are ground. The inference system (like the model theory) does not depend on any particular set of elementary queries and updates. Recall, however, that the version of $\mathcal{TD}$ presented in this paper has two kinds of elementary query, $b(\overline{x})$ and *b.empty*, and two kinds of elementary update, *ins.*$b(\overline{x})$ and *del.*$b(\overline{x})$, where b is a base predicate symbol.

Definition B.2 (Inference) $\Im$ is the following system of axioms and inference rules:

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

Inference Rules:

1. *Subroutines:* if $p \leftarrow \alpha$ is a ground instantiation of a rule in $\mathbf{P}$, then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash \psi'}{\mathbf{P}, \mathbf{D} \text{---} \vdash \psi}$$

where ψ' is obtained from ψ by replacing a hot occurrence of p by α.

2. *Elementary Queries:* if q is an elementary query that is true on $\mathbf{D}$, then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash \psi'}{\mathbf{P}, \mathbf{D} \text{---} \vdash \psi}$$ where ψ' is obtained from ψ by deleting a hot occurrence of q.

3. *Elementary Updates:* if u is an elementary update that transforms database $\mathbf{D}_1$ into $\mathbf{D}_2$, then

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{---} \vdash \psi'}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash \psi}$$ where ψ' is obtained from ψ by deleting a hot occurrence of u.

4. *Isolated Transactions:* if $\odot\alpha$ is a hot component of ψ, then

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

□

Inference system $\Im$ has a simple interpretation. The axioms describe the empty goal, which represents a transaction that does nothing and always succeeds. The four inference rules describe more complex transactions. Each rule involves two goals, ψ' and ψ, and gives conditions under which ψ succeeds, assuming that ψ' succeeds. For formal purposes, Definition B.2 often describes ψ' as the result of *removing* a hot component from ψ. However, an intuitive explanation is best given in terms of the inverse operation, where ψ is the result of *attaching* a hot component to ψ'. With this in mind, we can interpret the inference rules as follows:

- Rule 1 is a version of modus ponens. It replaces a subroutine definition, α, by its calling sequence, p.
- Rule 2 assumes that q is a test or condition that is satisfied by $\mathbf{D}$. The rule says that if a program succeeds from state $\mathbf{D}$, then we can attach q to the program as a hot component, and it will still succeed from $\mathbf{D}$.
- Rule 3 says that if a program succeeds from state $\mathbf{D}_2$, then we can attach update u to the program as a hot component, and it will now succeed from state $\mathbf{D}_1$. Because of this rule, database updates are an integral part of proving theorems in system $\Im$.
- Rule 4 says that sequential execution is isolated execution. In particular, if α and ψ' execute sequentially, then we can attach α to ψ' as a hot component that executes concurrently, but in isolation.

In any logical inference system, sequents (or theorems) are derived from axioms. Formally, a *logical deduction* (or *proof*) of a sequent, seq_n, is a series of sequents, $seq_0, seq_1, ..., seq_n$, where each seq_i is an axiom or is derived from earlier sequents by an inference rule. To reduce the search space, many proof theories employ specialized inference strategies. Two well-known strategies are *bottom-up* and *top-down* inference (also known as *forward-chaining* and *backward-chaining*, respectively). In bottom-up inference, a deduction is constructed by starting from the *first* sequent, seq_0, which is an axiom, and then applying an inference rule

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

Figure 2: A logical deduction in system $\Im$.

to derive the second sequent, seq_1, and then applying another inference rule to derive the third sequent, seq_2, etc. In top-down inference, a deduction is constructed by starting with the *last* sequent, seq_n, then applying an inference rule *in reverse* to derive the second-last sequent, seq_{n-1}, then the third-last, etc, until an axiom sequent is reached. In $\Im$, top-down and bottom-up inference have a special importance, since they correspond closely to transaction execution [20]. As in classical logic programming, the procedural semantics of $\mathcal{TD}$ is based on top-down inference.

Example B.3 (Logical Deduction) Suppose that rulebase **P** contains the following two rules:

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

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

The following theorem is one of the main results in [21]. It characterizes the inferences in $\Im$ model-theoretically, and it assures us that what is inferred is independent of how the system is applied, be it top-down, bottom-up, or some combination of both.

Theorem B.4 (Soundness and Completeness) *The sequent* $\mathbf{P}, \mathbf{D} \text{---} \vdash \psi$ *is derivable in system* $\Im$ *if and only if* $\mathbf{P}, \mathbf{D}\mathbf{D}' \models \psi$ *for some database* $\mathbf{D}'$.

According to Theorem B.4, we can use system $\Im$ to determine whether a transaction program will succeed from a given initial state. However, we can do even more. As Figure 2 suggests, a logical deduction can itself correspond to a successful execution, and by proving theorems, we can update the database and compute the final database state. To formalize this idea, we introduce a specialized notion of deduction that corresponds to program execution.

Definition B.5 (Executional Deduction) An *executional deduction* in $\Im$ is a series of sequents, $seq_0, seq_1, ..., seq_n$, in which seq_0 is an axiom and each seq_{i+1} is derived from seq_i by an inference rule. □

The following theorem is one of the main results in [21]. The procedural semantics outlined in Appendix C relies on this theorem, as does our prototype system [60, 78], which implements a top-down, depth-first search for executional deductions.

Theorem B.6 (Executional Soundness and Completeness) $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \psi$ *if and only if there is an executional deduction in* $\Im$ *whose last sequent is* $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash \psi$ *and whose first sequent is* $\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\,)$.

C APPENDIX: Concurrency and Process Interaction

Concurrency in $\mathcal{TD}$ has an interleaving semantics. Intuitively, a $\mathcal{TD}$ program consists of a number of concurrent processes, where each process produces a sequence of elementary database operations. By interleaving these sequences, we obtain a new sequence of operations, which can then be executed. The set of legal interleavings is determined by interactions between the processes. For instance, if one process writes data that another process must read, then the write operation must come *before* the read operation in an interleaved sequence. These interactions are specified by $\mathcal{TD}$ programs. Given such a program, the declarative semantics of $\mathcal{TD}$ determines the set of all legal interleavings, and the procedural semantics produces a correct execution. A correct execution is one that has the same effect on the database as a legal interleaving.

The *declarative* semantics of $\mathcal{TD}$ has been published elsewhere [19, 21], and is reviewed in Appendix A. This section focuses on the *procedural* semantics, *i.e.*, on the actual execution of $\mathcal{TD}$ programs. The procedural semantics is based on the proof system developed in Appendix B, in which updating the database is an integral part of proving theorems. It has also been implemented in a prototype [78, 60] on top of the XSB deductive database system [75].[12] This section illustrates the procedural semantics through a series of examples. Each example has been tested on the prototype, and behaves as described below.

To convey the right intuition, goals without concurrent conjunction (*e.g.*, $p \otimes q \otimes r$) are called *sequential processes*, or simply *processes*. In the procedural semantics, processes are executed from left to right, and as execution proceeds, elementary operations from different processes are interleaved. If a process tries to read data that is not yet available, or if some condition is not yet satisfied, then the process is suspended or delayed. If every process is suspended, then the execution is deadlocked, and some process is aborted, rolled back, and restarted if possible. These possibilities are illustrated in the examples below. Each example presents a $\mathcal{TD}$ program, describes its legal interleavings according to the declarative semantics, and then describes a correct execution according to the procedural semantics.

Example C.1 (Simple Interleaving) Consider the goal $(a \otimes b) \mid (p \otimes q)$, where a, b, p, q represent elementary updates. The goal thus consists of two concurrent processes, where

[12]XSB is a freely-distributable, GPL'd deductive database system. For more information, please visit the site `http://www.cs.sunysb.edu/~sbprolog/`

each process is a sequence of two elementary updates. There are six ways to interleave the two processes, each of which is legal: *abpq*, *apbq*, *apqb*, *pqab*, *paqb*, *pabq*. To execute the goal, each process is executed from left to right, and the updates are interleaved non-deterministically. □

Example C.2 (Isolation) Consider the goal $p \mid (a \otimes b \otimes \odot(c \otimes d))$, where p, a, b, c, d represent elementary updates. The elementary updates can be interleaved in five possible ways, but according to the declarative semantics, only four of these are legal: *pabcd*, *apbcd*, *abpcd*, *abcdp*. The fifth interleaving, *abcpd*, is illegal since it violates the isolation of the subtransaction $\odot(c \otimes d)$.[13] The procedural semantics generates one of the legal interleavings in a straightforward way: the process $a \otimes b \otimes \odot(c \otimes d)$ is executed from left to right; and p is executed at some non-deterministic point, though *not* during the execution of $\odot(c \otimes d)$. □

In the two previous examples, the transactions are relatively easy to execute since they involve updates only. The remaining examples also involve queries. The procedural semantics for such transactions is more complex since a query can act as a test or condition that must be satisfied before execution can continue (as in Examples 2.2 and 3.4). This means that (*i*) the legal interleavings depend on the database state, (*ii*) delay, deadlock and abort are possible, and (*iii*) processes within a transaction may interact and communicate.

Example C.3 (Process Delay) Consider the goal $(b \otimes ins.c) \mid (c \otimes ins.d) \mid (d \otimes ins.e)$, where b, c, d, e are base atoms. This goal has three processes. If the initial database contains the atom b, but not c or d, then according to the declarative semantics, there is only one legal interleaving of these processes: b, $ins.c$, c, $ins.d$, d, $ins.e$. Any other interleaving will cause query c or d (or both) to be unsatisfied. The procedural semantics generates this interleaving in a straightforward way. All three processes begin executing from left to right. However, processes $c \otimes ins.d$ and $d \otimes ins.e$ are immediately delayed, since atoms c and d are not in the database. After process $b \otimes ins.c$ has finished, atom c is in the database, so process $c \otimes ins.d$ continues. After it has finished, atom d is in the database, so process $d \otimes ins.e$ continues. □

In the previous example, an unsatisfied query causes the execution of a process to be delayed. Sometimes, the process may even be aborted and undone. As a trivial example, consider the transaction $del.p \otimes p$, which consists of a single process. During execution, the atom p is first deleted from the database, and then the query p is evaluated. Of course, the query is unsatisfied. Since there are no other processes to execute, the transaction is aborted, and the database is rolled back to its initial state. The next example illustrates abort and rollback in the presence of multiple, concurrent processes.

[13] As described earlier, an execution of the goal must have the *same effect* as one of the legal interleavings. Thus, if *abcd* and p do not access the same data items, then a DBMS can interleave p with *abcd* in any way that is chooses.

Example C.4 (Deadlock and Rollback) Consider the following goal:

$$(del.c \otimes \odot(b \otimes ins.c)) \mid (del.b \otimes \odot(c \otimes ins.b))$$

where b and c are distinct base atoms. The goal has two processes. If the initial database contains the atoms b and c, then according to the declarative semantics, these processes have two legal interleavings:

$$del.c,\ b,\ ins.c,\ del.b,\ c,\ ins.b \qquad\qquad del.b,\ c,\ ins.b,\ del.c,\ b,\ ins.c$$

In effect, one process executes completely, and then the other process executes completely. This is because each process deletes (but eventually restores) information that the other process needs. The procedural semantics generates a legal interleaving, but not always in a simple left-to-right fashion, since deadlock and rollback may be involved. To see this, suppose that $del.c$ and $del.b$ are executed first (in some order). This leaves $\odot(b \otimes ins.c)$ and $\odot(c \otimes ins.b)$ to be executed. But both these goals immediately fail, since b and c are no longer in the database. At this point, the execution is deadlocked since neither process can continue; so one process is aborted and rolled back. This changes the database so that the other process can execute to completion. The aborted process then restarts and executes to completion as well. $\Box$

As in most concurrent programming languages, $\mathcal{TD}$ programs may communicate and synchronize themselves. This is possible because one program can read what another program writes. The next example, which is a simplified version of Example 3.4, illustrates this idea.

Example C.5 (Communication and Synchronization) The rules below specify a program that spawns two subprocesses. The subprocesses execute concurrently and communicate to synchronize several tasks.

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

The first rule defines the top-level program, which immediately splits into two subprocesses, called $processA$ and $processB$. Each subprocess executes a sequence of three tasks, and uses the atoms $startB_2$ and $startA_3$ for communication. We assume that none of the tasks access these two atoms, and that neither atom is in the database initially.

In the declarative semantics, a legal interleaving of the two subprocesses must satisfy the following constraints: $ins.startB_2 < startB_2$ and $ins.startA_3 < startA_3$; that is, the update $ins.startB_2$ must execute before the query $startB_2$ is satisfied; and the update $ins.startA_3$ must execute before the query $startA_3$ is satisfied. It follows that a legal interleaving must also satisfy the following constraint: $taskA_1 < taskB_2 < taskA_3$; that is, $taskA_1$ must finish before $taskB_2$ begins, and $taskB_2$ must finish before $taskA_3$ begins.

In the procedural semantics, the two subprocess begin by executing $taskA_1$ and $taskB_1$ concurrently. When $processB$ completes $taskB_1$, it is delayed until $processA$ completes $taskA_1$ and inserts the atom $startB_2$ into the database. In this way, the two subprocesses communicate, and *processB* is synchronized with *processA*. Likewise, when *processA* completes $taskA_2$, it is delayed until *processB* completes $taskB_2$ and inserts the atom $startA_3$ into the database. In this way, the two subprocesses again communicate (in the reverse direction), and *processA* is synchronized with *processB*. The procedural semantics thus produces a legal interleaving of the two subprocesses, and a correct execution of the program. □

D APPENDIX: $\mathcal{TD}$ is Data Complete for RE

This appendix proves Theorem 5.2, which states that $\mathcal{TD}$ is data complete for RE. The RE upper-bound follows immediately from the inference system given in Appendix B, since for any logical inference system, the set of theorems derived by the system is in RE. The rest of the appendix proves the RE lower-bound.

Our proof strategy is to use $\mathcal{TD}$ to encode the computations of an arbitrary 2-stack machine [56]. A 2-stack machine is like a Turing machine except that instead of a work tape, it has two stacks. It also has a read-only input tape. At each computation step, the machine may push or pop each stack, and it may move the input head left or right. The action that is taken depends on the state of the machine's finite control, on the top-most symbol of each stack, and on the symbol beneath the input head. 2-stack machines can simulate arbitrary Turing machines, so *re*-hardness follows. Besides proving the RE upper-bound in Theorem 5.2, the machine encodings developed in this appendix are crucial to the expressibility proof in Appendix E. The encodings themselves are best understood in terms of the proof theory developed in Appendix B.

Formally, suppose that M is a 2-stack machine, and $\overline{s}$ is an input string for the machine. We shall encode $\overline{s}$ as a database $\mathbf{D}(\overline{s})$, and M as a rulebase $\mathbf{P}(M)$. The encoding has the following property:

$$\mathbf{P}(M), \mathbf{D}(\overline{s})\, \mathbf{D}' \models commit \qquad \textit{if and only if} \qquad \mathbf{D}' = \mathbf{D}(\overline{s}) \;\; \textit{and} \;\; M \; \textit{accepts} \; \overline{s} \tag{2}$$

where *commit* is a derived predicate symbol of arity zero. In addition, the rulebase $\mathbf{P}(M)$ is independent of the input $\overline{s}$, and that the database $\mathbf{D}(\overline{s})$ can be constructed from $\overline{s}$ in polynomial time (polynomial in the length of $\overline{s}$). RE-hardness follows immediately by letting M be a machine that recognizes an RE-complete language.

It only remains to construct $\mathbf{P}(M)$ and $\mathbf{D}(\overline{s})$. The latter construction is easy. If $\overline{s} = c_1 c_2 ... c_n$ is the input string, then $\mathbf{D}(\overline{s})$ is the following set of atoms:[14]

$$\begin{array}{c} first(1) \qquad next(1,2) \qquad next(2,3) \;\; \cdots \;\; next(n-1,n) \qquad last(n) \\ string(1,c_1) \qquad string(2,c_2) \qquad string(3,c_3) \;\; \cdots \;\; string(n,c_n) \end{array}$$

Here, the predicates *first*, *next* and *last* encode a counter from 1 to n, and using this, predicate *string* encodes the input string. Intuitively, the atom $string(i,c)$ means that character c is in position i of the input string. This database can easily be constructed in polynomial time (polynomial in n).

[14] Although these atoms contain numbers, this is simply a notational convenience, and the numbers are uninterpreted.

The construction of $\mathbf{P}(M)$ is more complex. We first decompose the problem into two parts by adding the following rule to $\mathbf{P}(M)$:

$$commit \leftarrow simulate \otimes restore \tag{3}$$

This rule defines *commit* to be a sequence of two subtransactions. The first subtransaction simulates the 2-stack machine, and the second restores the database to its initial state, $\mathbf{D}(\overline{s})$. Simulating the machine is the larger problem, and is the main subject of this appendix.

Machine Configurations:

A crucial element in the simulation is the representation of machine configurations. A configuration of a 2-stack machine is specified by the state of the finite control, and the contents of the two stacks. We store some of this information in the database, but we represent most of it as a concurrent goal of the form $stack_1 \mid stack_2 \mid control(q)$. The formula $control(q)$ is an atom that means that the finite control is in state q. The formulas $stack_1$ and $stack_2$ are serial goals that represent the contents of the two stacks.

To represent a stack, it is convenient to break it into two parts. If a stack contains the string $b_1 b_2 ... b_n$, then we refer to the symbol b_1 as the *top* of the stack, and to the string $b_2 ... b_n$ as the *body* of the stack. We also refer to the two stacks as Stack 1 and Stack 2, respectively. For $i \in \{1, 2\}$, we represent the top of Stack i as the atom $top_i(b_1)$, and store it in the database. We represent the body of Stack i as the following serial goal:

$$cell_i(b_2) \otimes cell_i(b_3) \otimes \cdots \otimes cell_i(b_n) \otimes cell_i(end) \otimes cell_i(end)$$

The atom $cell_i(end)$ marks the bottom of the stack. Having two copies of this atom is simply a technical convenience: when the stack is empty, one copy of *end* is in the database, and the other copy is in the serial goal, to prevent the goal from disappearing.

As an example of a machine configuration, suppose that the finite control is in state q, that Stack 1 contains the string $a_1 a_2 ... a_i$, and that Stack 2 contains the string $b_1 b_2 ... b_j$. This configuration is represented by the following concurrent goal:

$$\begin{array}{ll} & cell_1(a_2) \otimes cell_1(a_3) \otimes \cdots \otimes cell_1(a_i) \otimes cell_1(end) \otimes cell_1(end) \\ \mid & cell_2(b_2) \otimes cell_2(b_3) \otimes \cdots \otimes cell_2(b_j) \otimes cell_2(end) \otimes cell_2(end) \\ \mid & control(q) \end{array} \tag{4}$$

In addition, the atoms $top_1(a_1)$ and $top_2(b_1)$ are stored in the database.

The three serial goals making up a machine configuration can be thought of as three processes, a control process and two stack processes. In this view, when simulating a machine, the control process communicates with the two stack processes by sending messages to push or pop the stacks. The sending and receiving of messages is handled by rules (described below) that define the predicates $cell_1$, $cell_2$ and *control*. These rules unfold and execute when a process receives a message.

Initializing the Machine Configuration:

The machine simulation is best understood in terms of top-down inference. From this perspective, the top-level inference goal is the atom *simulate*. This atom is defined by the following rule, which initiates the machine simulation:

$$simulate \leftarrow init_1 \mid init_2 \mid control(q_0) \tag{5}$$

Here, q_0 is the initial state of the finite control of machine M, and the predicates $init_1$ and $init_2$ initialize the contents of the two stacks. Without loss of generality, we assume that the two stacks are initially empty, and that the first thing the machine does is to read the entire input string and push it onto Stack 2. (Intuitively, this stores the input string in working memory, eliminating the need to ever read the input tape again.) Setting up this initial configuration is the job of predicates $init_1$ and $init_2$.

The following rule defines $init_1$, which initializes Stack 1 to the empty stack:

$$init_1 \leftarrow ins.top_1(end) \otimes ins.ready_1 \otimes cell_1(end) \tag{6}$$

This rule first inserts the atom $top_1(end)$ into the database, to represent the top of the empty stack. It then inserts the atom $ready_1$ into the database, to signal the control process that Stack 1 is ready to be manipulated. The rule also creates the goal $cell_1(end)$, which represents the body of the empty stack.

The three rules below define $init_2$, which initialize Stack 2. They recursively read the input string $\overline{s}$ from the database and push it onto Stack 2, one symbol at a time.

$$\begin{aligned} init_2 &\leftarrow first(N) \otimes build(N) \otimes cell_2(end) \otimes cell_2(end) \\ build(N) &\leftarrow string(N,C) \otimes next(N,M) \otimes build(M) \otimes cell_2(C) \\ build(N) &\leftarrow string(N,C) \otimes last(N) \otimes ins.top_2(C) \otimes ins.ready_2 \end{aligned} \tag{7}$$

The first rule initializes the body of the stack by creating the goal $cell_2(end) \otimes cell_2(end)$. The second rule recursively scans the input string. At each recursive call, it reads a character, C, from the string, and pushes it onto the stack by creating the goal $cell_2(C)$. The third rule terminates the recursion when the last character of the input string is reached, and inserts it into the database as the atom $top_2(C)$, to represent the top of Stack 2. The third rule also adds the atom $ready_2$ to the database, to signal the control process that Stack 2 is ready to be manipulated.

The effect of the above rules can be summarized as follows. The top-level inference goal is the atom *simulate*, which is immediately reduced to the goal $init_1 \mid init_2 \mid control(q_0)$. This goal is further reduced as the two stacks are initialized. For instance, if $\overline{s} = abcde$ is the input string stored in the database, then the goal is reduced to the following:

$$\begin{aligned} & cell_1(end) \\ \mid\ & cell_2(d) \otimes cell_2(c) \otimes cell_2(b) \otimes cell_2(a) \otimes cell_2(end) \otimes cell_2(end) \\ \mid\ & control(q_0) \end{aligned}$$

In the process, the atoms $top_1(end)$, $top_2(e)$, $ready_1$, $ready_2$ are inserted into the database. Thus, Stack 1 is empty, Stack 2 contains the input string, and the control process has been signalled that both stacks are ready.

The Control Process:

The control process simulates the finite control of the 2-stack machine. It sends messages to the stack processes commanding them to push or pop their contents. It then waits until receiving messages from the stacks acknowledging that the commands have been carried out. This send-wait-receive cycle is repeated over-and-over again. Each cycle transforms the machine from one configuration to the next.

At the beginning of each cycle, the control process waits for the stacks to become ready (*i.e.*, to finish their previous commands). Stack i signals that it is ready by inserting the atom $ready_i$ into the database. When both stacks are ready, the control process examines the top of each stack (and its own internal state) to determine what commands to send to the stacks next. This is accomplished by the following two rules:

$$\begin{array}{rcl} control(Q) & \leftarrow & wait \otimes top_1(C_1) \otimes top_2(C_2) \otimes transform(Q, C_1, C_2) \\ wait & \leftarrow & ready_1 \otimes ready_2 \otimes del.ready_1 \otimes del.ready_2 \end{array}$$

The $wait$ predicate is a transaction that cannot execute until it receives a "ready" message from both stacks, *i.e.*, until the atoms $ready_1$ and $ready_2$ are in the database. When the $wait$ predicate executes, it reads and deletes the atoms $ready_1$ and $ready_2$ from the database (thereby consuming the "ready" messages).

The rule defining $control(Q)$ ultimately invokes the predicate $transform(Q, C_1, C_2)$, where Q is the internal state of the finite control, and C_i is the top element of Stack i. This predicate transforms the machine from one configuration to the next. It encodes each transition of the machine's finite control as a single rule. For instance, suppose the finite control has the following transition:

> If the control is in state q, the top of Stack 1 contains symbol b, and the top of Stack 2 contains symbol c; then pop Stack 1, push symbol d onto Stack 2, and change the control state to q'.

This transition is encoded as the following rule:

$$transform(q, b, c) \quad \leftarrow \quad ins.pop_1 \otimes ins.push_2(d) \otimes control(q')$$

This rule first inserts the atoms pop_1 and $push_2(d)$ into the database. These atoms act as messages to the stack processes, messages to pop Stack 1, and to push d onto Stack 2, respectively. The rule then invokes the control process recursively with the new state, q'. The control process then waits for the stacks to carry out the push and pop actions before beginning the next cycle.

The Stack Processes:

Recall that if Stack i stores the string $b_1 b_2 ... b_n$, then the top of the stack is represented by the atom $top_i(b_1)$ in the database, and the body of the stack is represented by the following serial goal:

$$cell_i(b_2) \otimes cell_i(b_3) \otimes \cdots \otimes cell_i(b_n) \otimes cell_i(end) \otimes cell_i(end) \tag{8}$$

To pop the stack, we must replace $top_i(b_1)$ by $top_i(b_2)$ in the database, and we must consume the atom $cell_i(b_2)$ from the front of the serial goal. The following two rules do exactly this:

$$\begin{aligned} cell_i(Z) &\leftarrow pop_i \otimes del.pop_i \otimes change_i(Y,Z) \otimes ins.ready_i \\ change_i(Y,Z) &\leftarrow top_i(Y) \otimes del.top_i(Y) \otimes ins.top_i(Z) \end{aligned}$$

The first rule does not execute until it receives a "pop" message from the control process, *i.e.*, until the atom pop_i is in the database. When it executes, the head of the rule consumes an atom of the form $cell_i(Z)$ from the front of the serial goal. The body of the rule then updates the database. The body first reads and deletes the atom pop_i (thereby consuming the "pop" message). It then executes the transaction $change(Y,X)$, which changes the top element of the stack from Y to Z. This completes the pop operation.

To push a symbol, b_0, onto the stack, we must replace $top_i(b_1)$ by $top_i(b_0)$ in the database, and we must add the atom $cell_i(b_1)$ to the front of the serial goal (8). This latter action is done indirectly by first consuming the atom $cell_i(b_2)$ from the front of the serial goal, and then attaching the atoms $cell_i(b_1)$ and $cell_i(b_2)$. The following rule does exactly this:

$$cell_i(Z) \leftarrow push_i(X) \otimes del.push_i(X) \otimes change_i(Y,X) \otimes ins.ready_i \otimes cell_i(Y) \otimes cell_i(Z)$$

This rule does not execute until it receives a "push" message from the control process, *i.e.*, until the atom $push_i(b_0)$ is in the database, and b_0 is bound to variable X. When it executes, the head of the rule consumes an atom of the form $cell_i(Z)$ from the front of the serial goal. The body of the rule then updates the database and creates more goals. The body first reads and deletes the atom $push_i(b_0)$ from the database (thereby consuming the "push" message). It then executes the transaction $change(Y,X)$, which changes the top element of the stack from Y to X. The body finally attaches two atoms to the front of the serial goal by creating the serial subgoal $cell_i(Y) \otimes cell_i(Z)$. This completes the push operation.

Terminating the Simulation and Restoring the Initial Database:

We must provide rules to terminate the simulation when the finite control enters an accepting state. This means that each of the three serial goals in configuration (4) must be reduced to the empty goal. Intuitively, this means terminating the control and stack processes.

We do this in two steps. In the first step, the control process broadcasts a halt message to the stacks, and then terminates. This is carried out by the following rule, which we add to the transaction base for each accepting control state, q_a:

$$control(q_a) \leftarrow wait \otimes ins.halt$$

This rule waits for the previous cycle to finish, and then inserts the atom *halt* into the database. Since this rule does not call the *control* predicate recursively, the control process is immediately reduced to the empty goal, and thus terminates. In the second step, the stack processes receive the halt message, and then terminate. This is carried out by the following two rules:

$$\begin{array}{rcl} cell_1(Z) & \leftarrow & halt \\ cell_2(Z) & \leftarrow & halt \end{array}$$

These rules have no effect until the atom *halt* is in the database, at which point they consume every atomic goal in the stack processes. This reduces each stack process to the empty goal, thereby terminating the process.

In this way, when the machine simulation reaches an accepting state, the machine configuration (4) is immediately reduced to the empty goal. Our machine simulation is thus a chain of inference that begins with the atom *simulate* and reaches the empty goal if and only if the machine enters an accepting state. Reaching the empty goal marks the end of a successful chain of inference. In fact, this is the only way that inference can succeed using the proof theory for Concurrent Transaction Logic developed in [21], since every axiom of the inference system involves the empty goal. We therefore have the following result:

$$\mathbf{P}(M), \mathbf{D}(\overline{s})\mathbf{D}'' \models simulate \quad \textit{for some state } \mathbf{D}'' \textit{ if and only if machine } M \textit{ accepts string } \overline{s}. \tag{9}$$

If the machine accepts its input, then $\mathbf{D}''$ is the state of the database at the end of the simulation. This state is not much different from the initial state, $\mathbf{D}(\overline{s})$. None of the atoms in $\mathbf{D}(\overline{s})$ are deleted by the simulation. Many atoms were added, but only temporarily. Some of these atoms represented the top elements of the stacks, and changed continually. Other atoms represented messages passed between concurrent processes: one process would write an atom, and another process would read and delete it. The end result is that the *simulate* transaction makes very few permanent changes to the database. In fact, its net effect is to leave the atom *halt* in the database (the last message), as well as two atoms of the form $top_1(a)$ and $top_2(b)$ (the top elements of the two stacks at the moment when the machine entered the accepting state). Thus, the the database is easily restored to its initial state by deleting these three atoms. The following rule does exactly this:

$$restore \leftarrow del.halt \otimes top_1(X) \otimes del.top_1(X) \otimes top_2(X) \otimes del.top_2(X) \tag{10}$$

We therefore have the following result:

$$\textit{if} \quad \mathbf{P}(M),\, \mathbf{D}(\overline{s})\,\mathbf{D}' \models simulate \otimes restore \quad \textit{then} \quad \mathbf{D}' = \mathbf{D}(\overline{s})$$

Together with statement (9) and rule (3), this proves property (2), and proves the RE-hardness of concurrent $\mathcal{TD}$.

E APPENDIX: $\mathcal{T}_{\mathcal{D}}$ is Expressively Complete for RE

This appendix uses the machine encodings developed in Appendix D to prove Theorem 5.5. That is, we prove the following result:

Theorem E.1 *Let $\mathcal{T}$ be a generic transaction of type $\overline{\alpha} \to \overline{\beta}$ in RE. If $\mathcal{T}$ preserves constants, then there is a rulebase $\mathbf{P}$ in concurrent $\mathcal{TD}$ such that*

$$\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models commit \quad \textit{if and only if} \quad \langle \mathbf{D}_1, \mathbf{D}_2 \rangle \in \mathcal{T}$$

for all databases $\mathbf{D}_1$ of type $\overline{\alpha}$, and all databases $\mathbf{D}_2$.

Let $\mathcal{T}$ be the database transaction in Theorem E.1. Since $\mathcal{T}$ is in RE, there is Turing machine that recognizes the set $\mathcal{T}$, *i.e.*, a machine that accepts a database pair $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$ if and only if it is an element of $\mathcal{T}$. There is therefore a 2-stack machine, M, that recognizes $\mathcal{T}$. Observe that since $\mathcal{T}$ is of type $\overline{\alpha} \to \overline{\beta}$ and constant preserving, the machine will accept a pair of databases $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$ only if $\mathbf{D}_1$ is of type $\overline{\alpha}$, and $\mathbf{D}_2$ is of type $\overline{\beta}$, and $Dom(\mathbf{D}_2) \subseteq Dom(\mathbf{D}_1)$. To simplify the presentation, we shall assume that $Dom(\mathbf{D}_2) = Dom(\mathbf{D}_1)$. It is not hard to generalize the proof from $=$ to $\subseteq$.

Appendix D showed how to construct a transaction base $\mathbf{P}(M)$ that simulates an arbitrary 2-stack machine, M. However, $\mathbf{P}(M)$ is *not* the rulebase $\mathbf{P}$ required in Theorem E.1. In particular, if $\overline{s}$ is the input string to machine M, then $\mathbf{P}(M)$ assumes that this string is encoded as a database, $\mathbf{D}(\overline{s})$. $\mathbf{D}(\overline{s})$ is therefore different from the database $\mathbf{D}_1$ in Theorem E.1, which is an arbitrary database of type $\overline{\alpha}$. We must provide rules which when given $\mathbf{D}_1$, will construct $\mathbf{D}(\overline{s})$. That is, we need rules that will encode the database as a string that can be used as input to our machine simulation. This kind of problem has been addressed by numerous authors in order to prove the expressiveness of query languages. We shall adapt the solution given in [16, 17]. The main new challenge is to deal with the non-determinism of transactions, and to generate the final database state.

In addition, to simplify the proof, we will allow machine M to have more than 2 stacks. Specifically, it will have $s+t+2$ stacks, where s is the number of relations in $\mathbf{D}_1$, and t is the number of relations in $\mathbf{D}_2$. One stack is used for each of the $s+t$ database relations, and two additional stacks are used as work space for the machine. The rules presented in Appendix D can be generalized from 2-stack machines to multi-stack machines in a straightforward way.

The rest of this appendix describes the rules that make up transaction base $\mathbf{P}$. Given a database $\mathbf{D}_1$ of type $\overline{\alpha}$, these rules carry out five main tasks. (i) They generate a linear order on the data domain, thus providing a counter. (ii) They non-deterministically generate a database $\mathbf{D}_2$ of type $\overline{\beta}$ with the same domain as $\mathbf{D}_1$. (iii) They encode each relation in $\mathbf{D}_1$ and $\mathbf{D}_2$ as a string. (iv) They load each string into a stack of machine M, and they simulate the machine. (v) If the machine accepts the input strings, then the database is cleaned up so that its final state is $\mathbf{D}_2$.

The encoding of database relations as strings in step (iii) depends on the linear order generated in step (i). That is, different linear orders will result in different strings. However, because machine M recognizes a *generic* transaction, acceptance of the input strings is independent of the particular linear order used to construct them. The result is that for any database $\mathbf{D}_2$ of type β with the same domain as $\mathbf{D}_1$,

$$\mathbf{P}, \mathbf{D}_1 \mathbf{D}_2 \models \mathit{commit} \quad \textit{if and only if} \quad \langle \mathbf{D}_1, \mathbf{D}_2 \rangle \in \mathcal{T} \tag{11}$$

Moreover, for any database $\mathbf{D}_2$ whose type is not $\overline{\beta}$ or whose domain is different from that of $\mathbf{D}_1$,

$$\mathbf{P}, \mathbf{D}_1 \mathbf{D}_2 \not\models \mathit{commit}$$

since $\mathbf{D}_2$ will not be generated in step (ii). Statement (11) thus holds for *all* databases $\mathbf{D}_2$, thereby proving Theorem E.1.

The rest of this appendix describes how the rules in $\mathbf{P}$ carry out their five main tasks. The entire process is controlled by the following rule:

$$\begin{aligned} commit \;\leftarrow\; & choose.order \otimes encode.db_1 \otimes erase.db_1 \otimes \\ & choose.db_2 \otimes encode.db_2 \otimes simulate \otimes restore.db_2 \end{aligned} \tag{12}$$

This rule expands the five main tasks into seven tasks, carried out by seven separate transactions.

1. Transaction *choose.order* non-deterministically chooses a linear order on the data domain and inserts it into the database.
2. Transaction $encode.db_1$ encodes each of the relations in database $\mathbf{D}_1$ as a string.
3. Transaction $erase.db_1$ erases the contents of each relation in $\mathbf{D}_1$, but leaves the data domain untouched.
4. Transaction $choose.db_2$ non-deterministically generates a database, $\mathbf{D}_2$, with the same domain as $\mathbf{D}_1$.
5. Transaction $encode.db_2$ encodes each of the relations in database $\mathbf{D}_2$ as a string.
6. Transaction *simulate* simulates machine M on input $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$. The transaction succeeds if and only if the machine accepts the input.
7. Transaction $restore.db_2$ restores the state of the database to $\mathbf{D}_2$.

The rest of this appendix describes the rules defining each of these seven transactions.

Generating a Linear Order:

This section defines the transaction *choose.order*, which non-deterministically chooses a linear order on the data domain, and inserts it into the database. This is accomplished by the rules below, which use the techniques of Examples **??** and **??**. In terms of top-down inference, these rules recursively choose elements from the data domain, one element at a time, and insert them into a growing linear order in the database. If the domain elements are chosen in the order $d_1, d_2, ..., d_n$, then the following atoms are inserted into the database:

$$first(d_1) \quad next(d_1, d_2) \quad next(d_2, d_3) \;\cdots\; next(d_{n-1}, d_n) \quad last(d_n). \tag{13}$$

The first rule begins the process by non-deterministically choosing a domain element, X, to be the first element in the linear order, and adding the atom $first(X)$ to the database. The second rule then chooses the remaining domain elements recursively. At each level of recursion, it non-deterministically chooses an element, Y, to be the next element in the linear order, and adds the atom $next(X, Y)$ to the database. As each domain element is chosen, it is removed (temporarily) from the domain, so that is will not be chosen again. This process continues until the domain is empty, at which point the third rule terminates the recursion, and adds the atom $last(X)$ to the database. Finally, as the recursion unwinds, the first and second rules re-insert each domain element back into the database, so that the domain is not destroyed by the computation.

$$
\begin{aligned}
choose.order &\leftarrow dom(X) \otimes ins.first(X) \otimes del.dom(X) \otimes loop(X) \otimes ins.dom(X) \\
loop(X) &\leftarrow dom(Y) \otimes ins.next(X,Y) \otimes del.dom(Y) \otimes loop(Y) \otimes ins.dom(Y) \\
loop(X) &\leftarrow dom.empty \otimes ins.last(X)
\end{aligned}
$$

The linear order generated by these rules allows us to count from 1 to n, where n is the size of the data domain. In addition, it induces a linear order on Dom^k, the space of k-tuples of domain elements. This allows us to count from 1 to n^k by using predicates of arity k. In particular, we can define predicates called $first_k(\overline{x})$, $last_k(\overline{x})$, and $next_k(\overline{x}, \overline{y})$, where $\overline{x}$ and $\overline{y}$ are k-tuples. Intuitively, $first_k(\overline{x})$ means that $\overline{x}$ represents the number 0, and $last_k(\overline{x})$ means that $\overline{x}$ represents the number $n^k - 1$, and $next_k(\overline{x}, \overline{y})$ means that $\overline{y}$ is the successor of $\overline{x}$ in the linear order. We can define such a counter for any k, which allows us to order the tuples in any database relation.

The rules below, adapted from [16], illustrate the idea for $k = 2$. In these rules, each pair (x, y) represents a number between 0 and $n^2 - 1$, where x and y represent digits in base n, and y is the low-order digit. Intutively, the first rule says that the number 0 is represented by $(0, 0)$, and the second rule says that $n^2 - 1$ is represented by $(n - 1, n - 1)$. The third rule says that adding 1 to (x, y) gives $(x, y + 1)$, as long as y has a successor. The last rule covers the case in which y does not have a successor: it says that adding 1 to $(x, n - 1)$ gives $(x + 1, 0)$, *i.e.*, it describes how to carry a 1 in base n.

$$
\begin{aligned}
first_2(X,Y) &\leftarrow first(X) \otimes first(Y) \\
last_2(X,Y) &\leftarrow last(X) \otimes last(Y) \\
next_2(X,Y_1,\ X,Y_2) &\leftarrow next(Y_1,Y_2) \\
next_2(X_1,Y_1,\ X_2,Y_2) &\leftarrow last(Y_1) \otimes first(Y_2) \otimes next(X_1,X_2)
\end{aligned}
$$

Encoding Base Predicates as Strings:

Recall that the initial database, $\mathbf{D}_1$, is represented by predicate symbols $p_1, p_2 ... p_s$, and the final database, $\mathbf{D}_2$, is represented by predicate symbols $q_1, q_2 ... q_t$, where any of the q_i may be the same as any of the p_i. We need to define transactions $encode.db_1$ and $encode.db_2$ that encode each of these relations as strings. At the top level, these two transactions are defined as follows:

$$
\begin{aligned}
encode.db_1 &\leftarrow encode.p_1 \otimes \cdots \otimes encode.p_s \\
encode.db_2 &\leftarrow encode.q_1 \otimes \cdots \otimes encode.q_t
\end{aligned}
$$

Here, the transaction $encode.p_i$ encodes relation p_i as a binary string. Likewise for transaction $encode.q_i$.

To encode a single relation as a binary string, suppose that p is a base relation of arity k. As described above, there are n^k different tuples of arity k, where n is the size of the data domain. We can therefore represent p as a binary string of length n^k, where each bit in the string represents a possible k-tuple. A bit is 1 if the corresponding tuple is in the database; otherwise, the bit is 0. To represent this string, we use the predicate $string(x_1, x_2, ..., x_k, c)$, where each x_i is a domain element, and c is 0 or 1. If we abbreviate $x_1, x_2 ... x_k$ by $\overline{x}$, then

$\overline{x}$ can be interpreted as a number between 0 and $n^k - 1$. Intuitively, the atom $string(\overline{x}, c)$ means the bit at position $\overline{x}$ has value c. Thus, if the atom $p(\overline{x})$ is in the database, then we generate the atom $string(\overline{x}, 1)$; otherwise, we generate the atom $string(\overline{x}, 0)$.

Using this representation, the rules below define the transaction *encode.p*, which encodes predicate p as a binary string. The encoding has two phases: (i) set every bit of the string to 0, and (ii) change a bit to 1 if the corresponding tuple is in the database. The first step uses the counter for k-tuples described above, and the second step uses the techniques described in Examples **??** and **??**.

$$encode.p \;\leftarrow\; zeros \otimes ones$$

<u>Set every bit to 0</u>

$$\begin{aligned}
zeros &\;\leftarrow\; first_k(\overline{X}) \otimes ins.string(\overline{X}, 0) \otimes loop(\overline{X}) \\
loop(\overline{X}) &\;\leftarrow\; next_k(\overline{X}, \overline{Y}) \otimes ins.string(\overline{Y}, 0) \otimes loop(\overline{Y}) \\
loop(\overline{X}) &\;\leftarrow\; last_k(\overline{X})
\end{aligned}$$

<u>Change some bits to 1</u>

$$\begin{aligned}
ones &\;\leftarrow\; p(\overline{X}) \otimes del.string(\overline{X}, 0) \otimes ins.string(\overline{X}, 1) \otimes del.p(\overline{X}) \otimes ones \otimes ins.p(\overline{X}) \\
ones &\;\leftarrow\; p.empty
\end{aligned}$$

The table below illustrates the effects of these rules for the special case when $k = 1$. It shows how a unary base predicate p is encoded as a binary string. For simplicity, the predicate *dom* is abbreviated as d, the predicate *string* is abbreviate as s, and other base predicates are not shown. The top row of the table shows that the data domain is $\{u, v, w, x, y, z\}$ and that the database stores the atoms $\{p(u),\ p(v),\ p(w)\}$. The next two rows show how predicate s is derived from predicate p. Note that the string encoded by s depends on how the data domain is ordered. If the domain order is $uvwxyz$, then the last row of the table represents the binary string 111000. On the other hand, if the domain order is $zyxwvu$, then the last row of the table represents the binary string 000111.

$\mathbf{D}_a$	$d(u)$	$d(v)$	$d(w)$	$d(x)$	$d(y)$	$d(z)$
	$p(u)$	$p(v)$	$p(w)$			
$\mathbf{D}_b$	$d(u)$	$d(v)$	$d(w)$	$d(x)$	$d(y)$	$d(z)$
	$p(u)$	$p(v)$	$p(w)$			
	$s(u,1)$	$s(v,1)$	$s(w,1)$	$s(x,1)$	$s(y,1)$	$s(z,1)$
$\mathbf{D}_c$	$d(u)$	$d(v)$	$d(w)$	$d(x)$	$d(y)$	$d(z)$
	$p(u)$	$p(v)$	$p(w)$			
	$s(u,1)$	$s(v,1)$	$s(w,1)$	$s(x,0)$	$s(y,0)$	$s(z,0)$

Formally, the following statements are all true:

$$\mathbf{P}, \mathbf{D}_a\mathbf{D}_b \models zeros \qquad \mathbf{P}, \mathbf{D}_b\mathbf{D}_c \models ones \qquad \mathbf{P}, \mathbf{D}_a\mathbf{D}_c \models encode.p$$

To encode predicates $p_1, ..., p_s$ and $q_1..., q_t$ as binary strings, we need $s + t$ versions of the above rules, one version for each encoded predicate. Each version of the rules has its own version of the predicate symbols involved, *i.e.*, $string_i$, $zeros_i$, $ones_i$ and $loop_i$, where $1 \leq i \leq s + t$.

Transforming the Database:

Transforming the database consists of two steps: erasing the old database relations $(p_1, ..., p_s)$, and generating new ones $(q_1, ..., q_t)$. These two steps are carried out by the transactions $erase.db_1$ and $choose.db_2$, respectively.

Transaction $erase.db_1$ is defined by $2s + 1$ rules. Here is the top-level rule:

$$erase.db_1 \leftarrow erase.p_1 \otimes erase.p_2 \otimes \cdots \otimes erase.p_s$$

Intuitively, $erase.p_i$ is a transaction that erases the contents of relation p_i from the database. It is defined by two rules, where $\overline{X}$ is an abbreviation for the variables $X_1, ..., X_{\alpha_i}$, and α_i is the arity of p_i.

$$\begin{array}{lll} erase.p_i & \leftarrow & p_i(\overline{X}) \otimes del.p_i(\overline{X}) \otimes erase.p_i \\ erase.p_i & \leftarrow & p_i.empty \end{array}$$

These rules recursively choose a tuple from p_i and delete it, continuing until p_i is empty.

Transaction $choose.db_2$ generates a new database non-deterministically. It is defined by $3t + 1$ rules. Here is the top-level rule:

$$choose.db_2 \leftarrow choose.q_1 \otimes choose.q_2 \otimes \cdots \otimes choose.q_t$$

Intuitively, $choose.q_i$ is a transaction that non-deterministically inserts a set of tuples into relation q_i. It is defined by the following three rules, where $\overline{X}$ is an abbreviation for the variables $X_1, X_2, ..., X_{\beta_i}$, and β_i is the arity of q_i:

$$\begin{array}{lll} choose.q_i & \leftarrow & choose.tuple_i(\overline{X}) \otimes ins.q_i(\overline{X}) \otimes choose.q_i \\ choose.q_i & \leftarrow & \end{array}$$

$$choose.tuple_i(X_1, X_2, ..., X_{\beta_i}) \leftarrow dom(X_1) \otimes dom(X_2) \otimes \cdots \otimes dom(X_{\beta_i})$$

The predicate $choose.tuple_i(\overline{x})$ is true if $\overline{x}$ is a tuple of arity β_i over the data domain. The first rule uses this predicate to non-deterministically choose a tuple $\overline{X}$, which it then inserts into relation q_i in the database. The rule then calls $choose.q_i$ recursively. It is important to note that the depth of recursion is non-deterministic. At any call to $choose.q_i$, either the first or second rule may be used, and the choice is non-deterministic. If the first rule is chosen, then another tuple is added to q_i. If the second rule is chosen, then the recursion is terminated. The result is that $choose_{q_i}$ has many execution paths, and that on each such path, a set of tuples from Dom^{β_i} is inserted into q_i. (The same tuple may be inserted more than once, but this does not create duplicates, since the database is a *set* of atoms.) In this way, $choose.q_i$ generates an extent for predicate q_i non-deterministically. Consequently, $choose.db_2$ generates an extent for database $\mathbf{D}_2$ non-deterministically.

Simulating the Multi-Stack Machine:

In Appendix D, we simulated a 2-stack machine with a $\mathcal{TD}$ program having three concurrent processes: one process for each stack, and one process for the machine's finite control. This idea is easily generalized to machines with arbitrarily many stacks. Recall that our machine has $s+t+2$ stacks: two stacks for work space, and $s+t$ stacks for the encoded predicates of $\mathbf{D}_1$ and $\mathbf{D}_2$. We can simulate this machine with a $\mathcal{TD}$ program having $s+t+3$ processes. Thus, instead of using rule (5) to initialize the machine, we use the following rule:

$$simulate \leftarrow init_1 \mid \cdots \mid init_{s+t} \mid init_1^w \mid init_2^w \mid control(q_0) \tag{14}$$

Here, transactions $init_1$, ..., $init_{s+t}$ load the base predicates $string_1$, ..., $string_{s+t}$ into $s+t$ different stacks. The rules defining these transactions are very similar to rules (7) in Appendix D. In addition, transactions $init_1^w$ and $init_2^w$ initialize the two empty work stacks. The rules defining these two transactions are very similar to rule (6) in Appendix D. Once the stacks have been initialized, transaction $control(q_0)$ begins the simulation of the multi-stack machine (as described in Appendix D) starting from state q_0, the initial state of the finite control. Transaction *simulate* succeeds if and only if the machine accepts the strings loaded into its stacks, *i.e.*, if and only if $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle \in \mathcal{T}$, as required by statement (11).

Restoring the Database:

Many of the transactions defined above add atoms to database $\mathbf{D}_2$, but none of them remove atoms. Consequently, after the machine simulation terminates, the database contains all the atoms in $\mathbf{D}_2$ plus some others. The "others" consist of those atoms used to encode $\mathbf{D}_1$ and $\mathbf{D}_2$ as strings, and atoms left over from the machine simulation. To restore the database to state $\mathbf{D}_2$, we must delete all these extra atoms. The following rule conveniently divides the process into two steps:

$$restore.db_2 \leftarrow restore_1 \otimes restore_2$$

Here, transaction $restore_1$ deletes the atoms left over from the machine simulation, as described in Appendix D. It can be defined by a single rule similar to rule (10). Transaction $restore_2$ deletes the atoms used to encode $\mathbf{D}_1$ and $\mathbf{D}_2$ as strings. It must therefore delete relations $string_1$, ..., $string_{s+t}$, and it must delete the linear order on the data domain, *i.e.*, atoms (13). This is accomplished by the following rules

$$\begin{aligned} restore_2 &\leftarrow delete.order \otimes delete.strings \\ delete.order &\leftarrow del.first(X) \otimes del.last(Y) \otimes erase.next \\ delete.strings &\leftarrow erase.string_1 \otimes \cdots \otimes erase.string_{s+t} \end{aligned}$$

Here, the definitions of transactions $erase.next$ and $erase.string_i$ are very similar to that of transaction $erase.p_i$, defined above, which erases relation p_i from the database.

This completes the definition of transaction *commit* in rule (12), and thus completes the proof of Theorem E.1.

F APPENDIX: Sequential $\mathcal{TD}$ is in EXPTIME

This appendix proves the EXPTIME upper-bound for sequential $\mathcal{TD}$ in Theorem 5.6. Recall that in sequential $\mathcal{TD}$, all goals have the form $q_1 \otimes ... \otimes q_n$, and all rules have the form $q_0 \leftarrow q_1 \otimes ... \otimes q_n$, where each q_i is an atomic formula. These formulas are called *serial goals* and *serial rules*, respectively. Our proof strategy is to develop a logical inference system for executional entailment, and then show that all theorems of the system can be derived in exponential time. The inference system is *sound* for all $\mathcal{TD}$ programs, but it is not *complete*. A key step in the proof is showing the system *is* complete for *sequential* $\mathcal{TD}$ programs.

We actually develop a series of three inference systems. We use the first system to establish soundness and completeness, we use the third system to establish the EXPTIME upper bound, and we use the second system to show that the other two systems are equivalent. Each inference system manipulates expressions of the form $\mathbf{P}, \overline{\mathbf{D}} \vdash \phi$, called *sequents*. Here, $\mathbf{P}$ is a set of rules, ϕ is a ground goal, and $\overline{\mathbf{D}}$ is an execution trace. The informal meaning of such a sequent is that $\overline{\mathbf{D}}$ is an execution trace of program ϕ. In the first inference system, $\overline{\mathbf{D}}$ is an arbitrary trace. This corresponds to non-isolated execution, *i.e.*, execution in an environment where other processes may execute concurrently. In the other two inference systems, $\overline{\mathbf{D}}$ is a singleton trace, consisting of a single database pair, $\mathbf{D}_1\mathbf{D}_2$. This corresponds to isolated execution, *i.e.*, an environment with no other concurrent processes. In sequential $\mathcal{TD}$, all executions are isolated, since there are no concurrent processes. Observe that sequents of the form $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash \phi$ are syntactically similar to executional entailments, *i.e.*, to expressions of the form $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$. Soundness and completeness establishes a semantic connection as well.

As in Appendix B, each inference system is a collection of axioms and inference rules. An axiom is a sequent, and an inference rule is a small set of sequents separated by a horizontal line. An inference rule has a simple interpretation: if the sequents above the line can be inferred, then so can the sequents below the line. Based on the axioms, an inference system uses the inference rules to derive more-and-more sequents. The three inference systems developed here have different inference rules, but the same axioms. These axioms are given in Definition F.1 below. The four kinds of axioms given in this definition correspond to the four restrictions on structures given in Definition A.2.

The three inference systems have different inference rules, but the same axioms. These axioms are given in Definition F.1 below. The four kinds of axioms given in this definition correspond to the four restrictions on structures given in Definition A.2.

Definition F.1 (Axioms)
$\mathcal{A}$ is the following set of sequents, for every base predicate symbol, r, and all databases $\mathbf{D}, \mathbf{D}_1, \mathbf{D}_2$ such that $dom(\mathbf{D}_2) \subseteq dom(\mathbf{D}_1)$:

1. $\mathbf{P}, \mathbf{DD} \vdash r(\overline{x}) \quad$ *if* $\quad r(\overline{x}) \in \mathbf{D}$
2. $\mathbf{P}, \mathbf{DD} \vdash r.empty \quad$ *if* $\quad \mathbf{D}$ contains no atoms of the form $r(\overline{x})$
3. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash ins.r(\overline{x}) \quad$ *if* $\quad \mathbf{D}_2 = \mathbf{D}_1 + \{r(\overline{x})\}$
4. $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash del.r(\overline{x}) \quad$ *if* $\quad \mathbf{D}_2 = \mathbf{D}_1 - \{r(\overline{x})\}$

□

F.1 Non-Isolated Execution

In this section, we develop the first inference system, denoted $\mathcal{I}^1$, whose theorems are statements about non-isolated execution. By constructing a canonical model, we show that the system is sound and complete for sequential $\mathcal{TD}$ programs.

Definition F.2 (System $\mathcal{I}^1$)
Inference system $\mathcal{I}^1$ consists of the axioms in $\mathcal{A}$ together with the following inference rules:

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

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

If $\mathbf{P}, \overline{\mathbf{D}}_1 \vdash \phi$ is derivable in system $\mathcal{I}^1$, and trace $\overline{\mathbf{D}}_1$ reduces to $\overline{\mathbf{D}}_2$, then we say that $\mathbf{P}, \overline{\mathbf{D}}_2 \vdash \odot\, \phi$ in system $\mathcal{I}^1$. □

In Definition F.2, each inference rule has a simple interpretation: if the sequents above the line can be inferred, then the sequent below the line can also be inferred. Based on the axiom sequents, system $\mathcal{I}^1$ applies the inference rules to derive more-and-more sequents. The rest of this subsection shows that the sequents derived in this fashion are sound and complete; *i.e.*, they correspond to executional entailments. For this purpose, it is convenient to generalize the notion of executional entailment given in Definition A.6. We prove soundness and completeness in terms of this generalized entailment, and then specialize the result to executional entailment.

Definition F.3 (Generalized Executional Entailment) Let ϕ be a goal, $\mathbf{P}$ be a set of rules, and $\overline{\mathbf{D}}$ be an execution trace. Then $\mathbf{P}, \overline{\mathbf{D}} \mid\models \phi$ if $\mathbf{M}, \overline{\mathbf{D}} \models \phi$ for every model $\mathbf{M}$ of $\mathbf{P}$. □

There is a simple relationship between the notions of entailment given in Definitions A.6 and F.3:

$$\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi \qquad \textit{if and only if} \qquad \mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \mid\models \odot\phi \tag{15}$$

Theorem F.4 (Generalized Soundness) *If* $\mathbf{P}, \overline{\mathbf{D}} \vdash \phi$ *is derivable in system* $\mathcal{I}^1$ *then* $\mathbf{P}, \overline{\mathbf{D}} \mid\models \phi$.

Proof: From the model theory in Appendix A.3, it is straightforward to show that each of the axioms and inference rules of system $\mathcal{I}^1$ is sound. For instance, $\mathbf{P}, \mathbf{DD} \mid\models r(\overline{x})$ if $r(\overline{x}) \in \mathbf{D}$. This shows that axiom 1 is sound. Likewise, if $\mathbf{P}, \overline{\mathbf{D}}_1 \mid\models \phi_1$ and $\mathbf{P}, \overline{\mathbf{D}}_2 \mid\models \phi_2$ then $\mathbf{P}, \overline{\mathbf{D}}_3 \mid\models \phi_1 \otimes \phi_2$, where $\overline{\mathbf{D}}_3$ is the concatenation of $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$. This shows that inference rule 2 is sound. □

To prove that system $\mathcal{I}^1$ is complete, we construct a canonical model of the rulebase, $\mathbf{P}$. This model, defined proof theoretically, provides the necessary link between inference and semantics.

Definition F.5 (Canonical Model) Given a rulebase, $\mathbf{P}$, the canonical model of $\mathbf{P}$ is the following mapping, $\mathbf{M_P}$, from traces to sets of ground atoms:

$$\mathbf{M_P}(\overline{\mathbf{D}}) \;=\; \{\, q \mid \; \mathbf{P}, \overline{\mathbf{D}} \vdash q \;\; \textit{is derivable in system } \mathcal{I}^1 \,\}$$

□

Because the axioms in Definition F.1 correspond exactly to the restrictions in Definition A.2, it is not hard to see that a canonical model is a valid structure. The following two lemmas establish basic results about canonical models, from which completeness follows.

Lemma F.6 *If* $\mathbf{P}$ *is a set of serial rules, and* ϕ *is a ground serial goal, then* $\mathbf{M_P}, \overline{\mathbf{D}} \models \phi$ *if and only if* $\mathbf{P}, \overline{\mathbf{D}} \vdash \phi$ *is derivable in system* $\mathcal{I}^1$.

Proof: ϕ has the form $q_1 \otimes \cdots \otimes q_n$, where each q_i is a ground atom, and $n \geq 1$. The proof is a straightforward induction on n, where the base case is established by Definition F.5. □

Lemma F.7 *If* $\mathbf{P}$ *is a set of serial rules, then* $\mathbf{M_P}$ *is a model of* $\mathbf{P}$.

Proof: Suppose that $q \leftarrow q_1 \otimes \cdots \otimes q_n$ is a ground instance of a rule in $\mathbf{P}$. It is enough to show that $\mathbf{M_P}, \overline{\mathbf{D}} \models q \leftarrow q_1 \otimes \cdots \otimes q_n$ for any trace, $\overline{\mathbf{D}}$. This is established as follows:

if	$\mathbf{M_P}, \overline{\mathbf{D}} \models q_1 \otimes \cdots \otimes q_n$	
then	$\mathbf{P}, \overline{\mathbf{D}} \vdash q_1 \otimes \cdots \otimes q_n$	by Lemma F.6
	$\mathbf{P}, \overline{\mathbf{D}} \vdash q$	by inference rule 1 of system $\mathcal{I}^1$
	$\mathbf{M_P}, \overline{\mathbf{D}} \vdash q$	by Lemma F.6.

□

Theorem F.8 (Generalized Completeness) *Suppose* $\mathbf{P}$ *is a set of serial rules, and* ϕ *is a ground serial goal. If* $\mathbf{P}, \overline{\mathbf{D}} \mid\models \phi$ *then* $\mathbf{P}, \overline{\mathbf{D}} \vdash \phi$ *is derivable in system* $\mathcal{I}^1$.

Proof: Suppose that $\mathbf{P}, \overline{\mathbf{D}} \mid\models \phi$. Then $\mathbf{M}, \overline{\mathbf{D}} \models \phi$ for every model, $\mathbf{M}$, of $\mathbf{P}$, by Definition F.3. In particular, $\mathbf{M_P}, \overline{\mathbf{D}} \models \phi$, by Lemma F.7. Hence $\mathbf{P}, \overline{\mathbf{D}} \vdash \phi$ is derivable in system $\mathcal{I}^1$, by Lemma F.6. □

Theorems F.4 and F.8 show that system $\mathcal{I}^1$ is sound and complete in terms of *generalized* executional entailment (Definition F.3). The following corollary specializes this result to the original notion of executional entailment (Definition A.6). It focuses on entailments that correspond to isolated executions.

Corollary F.9 (Soundness and Completeness of $\mathcal{I}^1$) *Suppose* $\mathbf{P}$ *is a set of serial rules, and* ϕ *is a ground serial goal. Then* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$ *if and only if* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash \odot\,\phi$ *in system* $\mathcal{I}^1$.

Proof:

	$\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$	
iff	$\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \mid\models \odot\phi$	by statement (15)
iff	$\mathbf{P}, \overline{\mathbf{D}} \mid\models \phi$	for some trace $\overline{\mathbf{D}}$ that reduces to $\mathbf{D}_1\mathbf{D}_2$, by Definitions A.3 and F.3
iff	$\mathbf{P}, \overline{\mathbf{D}} \vdash \phi$	is derivable in system $\mathcal{I}^1$, since $\mathcal{I}^1$ is sound and complete for $\mid\models$
iff	$\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash \odot\,\phi$	in system $\mathcal{I}^1$, by Definition F.2, since $\overline{\mathbf{D}}$ reduces to $\mathbf{D}_1\mathbf{D}_2$.

□

F.2 Isolated Execution

In this section, we develop a second inference system, denoted $\mathcal{I}^2$, whose theorems are statements about isolated execution. By proving that it is equivalent to system $\mathcal{I}^1$, we show that system $\mathcal{I}^2$ is sound and complete for sequential $\mathcal{TD}$ programs. We then develop a variant of $\mathcal{I}^2$, denoted $\mathcal{I}^3$, whose complexity is easy to establish (Section F.3).

Definition F.10 (System $\mathcal{I}^2$)
Inference System $\mathcal{I}^2$ consists of the axioms in $\mathcal{A}$ together with the following inference rules:

1. $$\frac{\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash \phi}{\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash q} \quad \textit{if} \quad q \leftarrow \phi \text{ is a ground instantiation of a rule in } \mathbf{P}.$$

2. $$\frac{\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash \phi_1 \qquad \mathbf{P}, \mathbf{D}_2\mathbf{D}_3 \vdash \phi_2}{\mathbf{P}, \mathbf{D}_1\mathbf{D}_3 \vdash \phi_1 \otimes \phi_2}$$

□

Theorem F.11 (Equivalence of $\mathcal{I}^1$ and $\mathcal{I}^2$) *Suppose* $\mathbf{P}$ *is a set of serial rules, and* ϕ *is a ground serial goal. Then* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash \phi$ *is derivable in system* $\mathcal{I}^2$ *if and only if* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash \odot\, \phi$ *in system* $\mathcal{I}^1$.

Proof: We must show that the sequent $\mathbf{P}, \mathbf{D}_0\mathbf{D}_n \vdash \phi$ is derivable in system $\mathcal{I}^2$ if and only if some sequent of the form $\mathbf{P}, \mathbf{D}_0\mathbf{D}_1, \mathbf{D}_1\mathbf{D}_2, \mathbf{D}_2\mathbf{D}_3, ..., \mathbf{D}_{n-1}\mathbf{D}_n \vdash \phi$ is derivable in system $\mathcal{I}^1$. The proof is a straightforward induction over the length of derivations. For instance, the sequents $\mathbf{P}, \{\,\}\{b\} \vdash \textit{ins.b}$ and $\mathbf{P}, \{b\}\{bc\} \vdash \textit{ins.c}$ are both derivable in systems $\mathcal{I}_1$ and $\mathcal{I}_2$, since they are axioms. Thus, the sequent $\mathbf{P}, \{\,\}\{b\}, \{b\}\{bc\} \vdash \textit{ins.b} \otimes \textit{ins.c}$ is derivable by inference rule 2 of system $\mathcal{I}^1$. Likewise, the sequent $\mathbf{P}, \{\,\}\{bc\} \vdash \textit{ins.b} \otimes \textit{ins.c}$ is derivable by inference rule 2 of system $\mathcal{I}^2$. □

Definition F.12 (System $\mathcal{I}^3$) Inference system $\mathcal{I}^3$ consists of the axioms in $\mathcal{A}$ together with the following inference rule, where $q \leftarrow q_1 \otimes q_2 \otimes \cdots \otimes q_n$ is a ground instantiation of a rule in $\mathbf{P}$:

$$\frac{\mathbf{P}, \mathbf{D}_0\mathbf{D}_1 \vdash q_1 \qquad \mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash q_2 \quad \cdots \quad \mathbf{P}, \mathbf{D}_{n-1}\mathbf{D}_n \vdash q_n}{\mathbf{P}, \mathbf{D}_0\mathbf{D}_n \vdash q}$$

□

Theorem F.13 (Equivalence of $\mathcal{I}^2$ and $\mathcal{I}^3$) *Suppose* $\mathbf{P}$ *is a set of serial rules, and* q *is a ground atomic formula. Then* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash q$ *is derivable in system* $\mathcal{I}^3$ *if and only if it is derivable in system* $\mathcal{I}^2$.

Proof: A straightforward induction over the length of derivations. □

The following corollary is the basis for the EXPTIME upper bound proved below. The corollary follows immediately from the soundness and completeness of system $\mathcal{I}^1$ (Corollary F.9), and from the equivalence of systems $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{I}_3$ (Theorems F.11 and F.13).

Corollary F.14 (Soundness and Completeness of $\mathcal{I}^3$) *Suppose* $\mathbf{P}$ *is a set of serial rules, and* q *is a ground atomic formula. Then* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models q$ *if only if* $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash q$ *is derivable in system* $\mathcal{I}^3$.

F.3 EXPTIME Upper Bound

We wish to determine whether $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \phi$. We show that this can be done in exponential time (exponential in the size of $\mathbf{D}_1$) by deriving theorems in system $\mathcal{I}^3$. System $\mathcal{I}^3$ assumes that ϕ is a ground atomic formula, but this causes no loss of generality, since we can always add a rule of the form $q \leftarrow \phi$ to $\mathbf{P}$.

Our proof is based on a bottom-up iterative procedure that starts with the axioms sequents, and then applies the inference rule of system $\mathcal{I}^3$ repeatedly to derive more sequents. The only complication is that system $\mathcal{I}^3$ has infinitely many theorems, and bottom-up inference will try to derive all of them. For instance, $\mathbf{P}, \{\}\{p(c)\} \vdash ins.p(c)$ is a theorem for every predicate symbol, p, and every constant symbol, c. However, to derive a *particular* sequent, we do not need to consider *every* predicate and constant symbol. This is because $\mathcal{TD}$ is safe: it does not create new constant or predicate symbols during program execution. Formally, suppose we are trying to derive the sequent $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash q$. Let *dom* be the set of constant symbols appearing in this sequent, and let *pred* be the set of predicate symbols appearing in the sequent, including any predicate symbol, p, inside expressions of form *ins.p*, *del.p* or *p.empty*. In trying to derive the sequent, we may have to derive many other sequents as intermediate results, but we need only derive sequents built from the symbols in *pred* and *dom*. There are exponentially many sequents of this form (exponential in the size of *dom*). To see this, let $\mathcal{H}$ be the Herbrand base generated from *dom* and *pred*, *i.e.*, the set of ground atomic formulas built from the constant symbols in *dom* and the predicate symbols in *pred*. The size of this set is polynomial in the size of *dom*. Our derivations need only contain sequents of the form $\mathbf{P}, \mathbf{D}_3\mathbf{D}_4 \vdash q$ where $\mathbf{D}_i \subseteq \mathcal{H}$ and $q \in \mathcal{H}$. There are polynomially many choices for q, and exponentially many choices for each $\mathbf{D}_i$. Hence, there are exponentially many sequents.

System $\mathcal{I}^3$ uses the ground instances of rules in $\mathbf{P}$. There are infinitely many such instances, but we need only consider a finite subset of them: those in which rule variables are instantiated by constants in *dom*. Let $inst(\mathbf{P})$ be the set of all such instances. The size of this set is polynomial in the size of *dom*. Using this set, the following algorithm computes all the sequents built from *dom* and *pred* that are derivable in system $\mathcal{I}^3$.

S := the set of all axiom sequents built from *dom* and *pred*
(1) do until S no longer changes
(2)   do for each rule $q \leftarrow q_1 \otimes \cdots \otimes q_k$ in $inst(\mathbf{P})$
(3)   do for each set of $k+1$ databases $\{\mathbf{D}_0, \mathbf{D}_1, ..., \mathbf{D}_k\}$
         if S contains the following sequents:
            $\mathbf{P}, \mathbf{D}_0\mathbf{D}_1 \vdash q_1$
            $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \vdash q_2$
               $\cdots$
            $\mathbf{P}, \mathbf{D}_{k-1}\mathbf{D}_k \vdash q_k$
         then add the following sequent to S:
            $\mathbf{P}, \mathbf{D}_0\mathbf{D}_k \vdash q$

The running time of this algorithm is exponential time in the size of *dom*. To see this, first note that S is a set of sequents built from *dom* and *pred*. It therefore has at most exponential size at all times during execution. Moreover,

- S grows monotonically during execution, and never shrinks. Thus, each iteration of loop (1) (except the last) adds at least one sequent to S. Thus, there are at most exponentially many iterations of loop (1).
- During each iteration of loop (1), there are at most polynomially many iterations of loop (2), since $inst(\mathbf{P})$ has polynomial size. Thus, there are exponentially many iterations of loop (2) altogether.
- in loop (3), k is bounded above by a constant: the number of atoms in the largest rule in $\mathbf{P}$.
- In loop (3), there are exponentially many choices for the set $\{\mathbf{D}_0, \mathbf{D}_1, ..., \mathbf{D}_k\}$, since there are exponentially many choices for each $\mathbf{D}_i$. Thus, for each iteration of loop (2), there are exponentially many iterations of loop (3). Thus, there are exponentially many iterations of loop (3) altogether.
- Each iteration of loop (3) takes exponential time, since S has exponential size.

In sum, there are exponentially many iterations of loop (3), each taking exponential time, so the total execution time of the algorithm is exponential. Theorems of system $\mathcal{I}^3$ can thus be derived in exponential time. Hence, by Corollary F.14, executional entailments can be verified in exponential time. This proves the EXPTIME upper-bound in Theorem 5.6.

G APPENDIX: Sequential $\mathcal{TD}$ is EXPTIME-Hard

This appendix proves the EXPTIME lower-bound for sequential $\mathcal{TD}$ in Theorem 5.6. The main idea is to use sequential $\mathcal{TD}$ to simulate an alternating Turing machine. Like a non-deterministic machine, an alternating machine may have many possible transitions at any point in its computation, and it may require that *some* of these transitions be successful. In addition, an alternating machine may require that *all* the transitions be successful. The alternation between *all* and *some* during a computation gives these machines more power than ordinary non-deterministic machines [34]. For instance, alternating Turing machines that use polynomial space (APSPACE machines) have the same computational power as deterministic Turing machines that use exponential time (EXPTIME machines). Formally, APSPACE = EXPTIME [34]. We shall simulate arbitrary APSPACE machines in sequential $\mathcal{TD}$. Besides proving the EXPTIME lower-bound in Theorem 5.6, the machine encodings developed in this appendix are crucial to the expressibility proof in Appendix H. The encoding itself is a simple adaptation of an encoding developed in [16, 17] for a logical language called *Hypothetical Datalog*. The main difficulty is ensuring that the adaptation is correct.

Formally, suppose that M is a one-tape APSPACE machine, and $\overline{s}$ is an input string for the machine. We shall encode $\overline{s}$ as a database $\mathbf{D}(\overline{s})$, and M as a rulebase $\mathbf{P}(M)$. The encoding has the following property:

$$\mathbf{P}(M), \mathbf{D}(\overline{s})\, \mathbf{D}' \models accept \qquad \textit{if and only if} \qquad \mathbf{D}' = \mathbf{D}(\overline{s}) \;\; \textit{and} \;\; M \textit{ accepts } \overline{s} \tag{16}$$

where *accept* is a derived predicate symbol of arity zero. In addition, the rulebase $\mathbf{P}(M)$ is independent of the input $\overline{s}$, and that the database $\mathbf{D}(\overline{s})$ can be constructed from $\overline{s}$ in polynomial time (polynomial in the length of $\overline{s}$). EXPTIME-hardness follows immediately by letting M be a machine that recognizes an EXPTIME-complete language.[15] The rest of this section sketches the construction of $\mathbf{P}(M)$ and $\mathbf{D}(\overline{s})$.

The basic features of the construction are easy to represent in sequential $\mathcal{TD}$. The machine tape is represented as a set of atoms in the database, with one atom for each tape cell. Tape reads and writes are simulated as database queries and updates, respectively. Since the data domain and the database schema are fixed, the database size is at most polynomial in the size of the data domain. We can therefore represent machine tapes of at most polynomial size. For this reason, it is not difficult to simulate deterministic PSPACE machines in sequential $\mathcal{TD}$. Indeed, this idea has been applied to many safe update languages to show that their data complexity is complete for PSPACE (*e.g.*, [32, 5, 4]). Simulating an *alternating* machine is more complex, since at each point of the computation, the machine may have to follow *one* or *all* of the state transitions. Simulating *one* transition is easy, since $\mathcal{TD}$ can choose a transition non-deterministically. The main challenge is to simulate *all* the transitions.

Following all the transitions of an alternating machine effectively creates a computation *tree*. (For alternating PSPACE machines, this tree can have exponential depth, and double-exponential size.) Each branch of this tree represents a serial computation that must be executed. Normally, one thinks of these branches as being executed in parallel. However, it is easiest to imagine that our simulation program executes them serially, one branch at a time. To see this, consider the root of a successful computation tree. The root represents the start of a computation, so the machine tape is in its initial state, and the simulation program has many branches that it must execute. The program chooses a branch, simulates it to completion, and then restores the machine tape to its initial state. The program then chooses another branch, simulates it to completion, and again restores the machine tape to its initial state. The program continues in this way until every computation path in the tree has been simulated. A similar process occurs not just at the root, but at every node of the computation tree. Thus, in addition to simulating actual computation steps, the program is continually restoring the machine tape to earlier states. Since the tape is represented as a database, the program is continually restoring the *database* to earlier states. This is exactly what the language Hypothetical Datalog was designed to do, which is why it is able to simulate alternating PSPACE machines [16, 17]. We show that the same simulation can be done in sequential $\mathcal{TD}$. The main difference is that Hypothetical Datalog rolls back database updates automatically, while sequential $\mathcal{TD}$ uses explicit compensating transactions.

Representing the Tape as a Database:

Our representation of the machine is exactly the same as in [16, 17]. We outline this representation here, because we will need to refer to it later on.

Since M is a one-tape alternating Turing machine that runs in polynomial space, there is an integer k such that M uses at most n^k tape cells, where n is the length of the input

[15] See [34] for examples of such machines.

string, $\overline{s}$. To represent these positions, we put the following atoms in the database:[16]

$$first(1) \qquad next(1,2) \qquad next(2,3) \quad \cdots \quad next(n^k-1,n^k) \qquad last(n^k)$$

These atoms encode a counter from 1 to n^k. Using this counter, we represent the contents of the machine tape as a base predicate. Intuitively, the atom $cell(i,c)$ means that the tape cell at position i contains symbol c. Initially, the input string $\overline{s}$ is stored on the first n tape cells, and the remaining cells are blank. Thus, if $\overline{s} = c_1c_2...c_n$, then the following atoms are in the database $\mathbf{D}(\overline{s})$:

$$\begin{array}{l} cell(1,c_1) \qquad cell(2,c_2) \quad \cdots \quad cell(n,c_n) \\ cell(n+1,b) \qquad cell(n+2,b) \quad \cdots \quad cell(n^k,b) \end{array}$$

where b is the symbol for a blank. This database can easily be constructed in polynomial time (polynomial in n).

Representing the Machine as a Rulebase:

Our construction of the rulebase $\mathbf{P}(M)$ is a simple adaptation of a construction given in [16, 17] for Hypothetical Datalog. To adapt this construction, we show how programs in Hypothetical Datalog can be transformed into programs in $\mathcal{TD}$. The transformation is not perfect, since it does not preserve the semantics of *all* Hypothetical Datalog programs. However, it *does* preserve the semantics of our machine simulations, which is all we need. Before describing the transformation, we provide a brief overview of the source language.

Hypothetical Datalog is an extension of classical Datalog in which users can express hypothetical queries and updates. Informally, this means that a user can update the database temporarily (or hypothetically) before asking a query. The update is temporary since after query evaluation, the database is automatically restored to its initial state. To express this idea, Hypothetical Datalog includes two modalities, $q[add:b]$ and $q[del:b]$, where q and b are atomic formulas. Intuitively, $q[add:b]$ means, "q would be true if b were inserted into the database." Likewise, $q[del:b]$ means, "q would be true if b were deleted from the database." A *hypothetical rule* is a formula of the form $p \leftarrow \phi_1, \phi_2, ..., \phi_k$ where p is an atomic formula, and each ϕ_i is either an atomic formula or a hypothetical modality. Thus, a rule body may contain hypothetical queries, as well as ordinary queries. A finite set of hypothetical rules is called a hypothetical rulebase. Finally, the expression $\mathbf{R}, \mathbf{D} \vdash p$ intuitively means that atom p can be inferred from hypothetical rulebase $\mathbf{R}$ and database $\mathbf{D}$. The precise definition of inference, as well as a logical semantics, is developed in [18].

In [16], it is shown that the data complexity of Hypothetical Datalog is complete for EXPTIME. The proof shows that any alternating PSPACE machine, M, can be simulated in the language. Formally, M can be encoded as a hypothetical rulebase, $\mathbf{R}(M)$, such that for any input string, $\overline{s}$,

$$\mathbf{R}(M), \mathbf{D}(\overline{s}) \vdash accept \qquad \textit{if and only if} \qquad M \textit{ accepts } \overline{s}$$

[16]Although these atoms contain numbers, this is simply a notational convenience, and the numbers are uninterpreted.

where *accept* is a derived predicate symbol of arity zero, and $\mathbf{D}(\overline{s})$ is the database described above. Since $\mathbf{R}(M)$ is independent of $\overline{s}$, it immediately follows that the data complexity of Hypothetical Datalog is APSPACE-hard, and thus EXPTIME-hard.

We shall transform the hypothetical rulebase $\mathbf{R}(M)$ into a rulebase, $\mathbf{P}(M)$, in sequential $\mathcal{TD}$. The transformation is fairly direct, and each hypothetical rule is translated into a $\mathcal{TD}$ rule in a straightforward way. To do this, suppose that q and a are atomic formulas. Then, the modality $q[add : a]$ in Hypothetical Datalog is translated into the serial goal $ins.a \otimes q \otimes del.a$ in Transaction Datalog. Note that the query q is followed by an explicit delete, to restore the database to its original state. Likewise, the modality $q[del : a]$ is translated into the serial goal $del.a \otimes q \otimes ins.a$. Atomic formulas are unaffected by the translation, so the translation of q is q. Finally, the rule $p \leftarrow q_1, q_2, ..., q_k$ in Hypothetical Datalog is translated into the serial rule $p \leftarrow q'_1 \otimes q'_2 \otimes \cdots \otimes q'_k$ in Transaction Datalog, where q'_i is the translation of q_i, as just described. For example, the following hypothetical rule:

$$p(X) \;\leftarrow\; q(X,Y),\; r(Y)[add : a(Y)],\; s(Y)[del : b(Y)]$$

is translated into the following serial rule in sequential $\mathcal{TD}$:

$$p(X) \;\leftarrow\; q(X,Y) \otimes ins.a(Y) \otimes r(Y) \otimes del.a(Y) \otimes del.b(Y) \otimes s(Y) \otimes ins.b(Y)$$

This transformation requires some qualification, however, since it is only correct under certain conditions. For instance, suppose that database $\mathbf{D}$ does not contain the atom b. Then, deleting b from $\mathbf{D}$ has no effect, since $\mathbf{D} - \{b\} = \mathbf{D}$. In this case, the modality $q[del : b]$ is effectively the same as the atom q. However, our translation of the modality does *not* have this property. That is, the serial goal $del.b \otimes q \otimes ins.b$ is not the same as q. Instead, it is the same as $q \otimes ins.b$. In this case, the serial goal changes the database permanently, not temporarily, by adding b to the database after query q has been processed. Likewise, if $\mathbf{D}$ *does* contain the atom b, then the serial goal $ins.b \otimes q \otimes del.b$ changes the database by permanently deleting b, while the modality $q[add : b]$ does not.

Fortunately, this difference does not arise during our Turing-machine simulations. That is, the rules and the databases of the simulation are such that each hypothetical insertion and each hypothetical deletion actually changes the database every time it is executed. This is because the database represents a machine tape, and every database update represents a tape write. For instance, suppose that the tape cell at position i contains symbol b, and that the tape head overwrites it with symbol c. Thus, initially, the atom $cell(i, b)$ is in the database. To simulate the write operation, this atom is first deleted from the database, and then then atom $cell(i, c)$ is inserted. Thus, each tape write is simulated by a series of two database updates, each of which changes the database. As a special case, if the machine were to overwite the symbol b with b, then there would be two equal but opposite changes to the database: first the atom $cell(i, b)$ would be deleted, and then it would be re-inserted by a separate update. Thus, during a machine simulation, every insert and every delete actually changes the databases. Thus, our translation of $\mathbf{R}(M)$ into $\mathbf{P}(M)$ preserves the semantics of Hypothetical Datalog. Moreover, since every update is eventually undone, the database is left in its original state, $\mathbf{D}(\overline{s})$, when the simulation is finished. This establishes property (16), and proves the EXPTIME-hardness of sequential $\mathcal{TD}$.

H APPENDIX: Sequential $\mathcal{TD}$ is Expressively Complete for EXPTIME

This appendix uses the machine encodings developed in Appendix G to prove Theorem 5.7. Specifically, we prove the following result:

Theorem H.1 *Let $\mathcal{T}$ be a generic transaction of type $\overline{\alpha} \to \overline{\beta}$ in EXPTIME. If $\mathcal{T}$ preserves constants, then there is a rulebase* **P** *in sequential $\mathcal{TD}$ such that*

$$\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \models \textit{commit} \quad \textit{if and only if} \quad \langle \mathbf{D}_1, \mathbf{D}_2 \rangle \in \mathcal{T}$$

for all databases $\mathbf{D}_1$ of type $\overline{\alpha}$, and all databases $\mathbf{D}_2$.

Let $\mathcal{T}$ be the database transaction in Theorem H.1. Since $\mathcal{T}$ is in EXPTIME, there is an EXPTIME machine that recognizes the set $\mathcal{T}$, *i.e.*, a machine that accepts a database pair $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$ if and only if it is an element of $\mathcal{T}$. There is therefore an alternating PSPACE machine, M, that recognizes $\mathcal{T}$. To simplify the proof, we allow machine M to have more than one tape. Specifically, we assume that M has one work tape and $s + t$ input tapes, where s is the number of relations in $\mathbf{D}_1$, and t is the number of relations in $\mathbf{D}_2$. Each input tape is read-only and stores a single database relation. The machine encoding developed in Appendix G can be generalized from one-tape machines to multi-tape machines in a straightforward way.

The rules in **P** are similar to those in Appendix E. The main difference is that **P** uses the machine encodings developed in Appendix G, instead of those developed in Appendix D. As in Appendix D, we use tuples of domain elements to represent numbers of polynomial size. In particular, if $\overline{x}$ is a k-tuple, then it represents a number between 1 and n^k, where n is the size of the data domain. To count with these numbers, we again define the predicates $first_k(\overline{x})$, $next_k(\overline{x}, \overline{y})$ and $last_k(\overline{x})$. With these counters, we can represent a machine tape by a set of atoms in the database. In particular, we represent machine tape i by atoms of the form $cell_i(\overline{x}, c)$, where c represents a tape symbol, and $\overline{x}$ is a tuple representing a number. Intuitively, this atom means that the tape cell at position $\overline{x}$ contains symbol c.

In Appendix E, the transaction $encode.db_1$ encoded each relation in database $\mathbf{D}_1$ as a string. Likewise for $encode.db_2$. With virtually no change, we can use these two transactions to encode each database relation as a machine tape. This is possible for two reasons. First, the representation of machine tapes used here is virtually identical to the representation of strings used in Appendix E. Second, the database relations are stored on *read-only* input tapes. Thus, our representation of an input tape does not have to allow for movement of the tape head past the end of the string.

The encoding and initialization of the work tape is just slightly more complicated. Since machine M runs in alternating polynomial time, it uses at most n^k cells of its work tape, for some k. Positions on the work tape can therefore represented by k-tuples. We use the predicate symbol $cell_0$ to represent the tape. The rules below initialize the work tape, by generating a sequence of n^k tape cells, each containing a 0:

$$\begin{array}{rcl} init & \leftarrow & first_k(\overline{X}) \otimes ins.cell_0(\overline{X}, 0) \otimes loop(\overline{X}) \\ loop(\overline{X}) & \leftarrow & next_k(\overline{X}, \overline{Y}) \otimes ins.cell_0(\overline{Y}, 0) \otimes loop(\overline{Y}) \\ loop(\overline{X}) & \leftarrow & last_k(\overline{X}) \end{array}$$

Finally, we need to modify the definition of *simulate* in rule (14). This rule—the only non-serial rule in Appendix E—initializes the machine tapes and starts the machine simulation. We replace it by the following serial rule:

$$simulate \leftarrow init \otimes accept$$

Here, *accept* is a transaction that simulates the execution of machine M, as described in Appendix G. This rule therefore initializes the machine's work tape, and then starts the simulation.

This completes the proof of Theorem H.1, and thus of Theorem 5.7.