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@cs.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 their computational properties 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.

1 Introduction

The management of workflows and business processes is a ubiquitous task faced by many organizations in a wide range of industries [22, 26, 31]. 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" [22]. 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.

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

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 [15, 13]. 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 [24, 8]. Such dependencies are not accounted for in the classical theory of transaction management [7]. The result is that "business process management cannot be handled by means of conventional database techniques alone" [22]. For this reason, there has been considerable research on advanced transaction models [24] and their application to transactional workflow [31].

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, many kinds of Petri net, as well as temporal logic, state charts, concurrent transition systems, and concurrent logic programming (*e.g.*, [36, 29, 37, 27]). 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.*, [19, 34]). 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 [37]). 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 [10, 8]. $\mathcal{TD}$ 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 [22], 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 [11, 10]. $\mathcal{TD}$ also has many features of advanced transaction models, including subtransaction hierarchies, relaxed ACID requirements, and fine-grained control over abort and rollback [8]. 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 [5, 21, 42]. 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 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 [3, 4]. 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 [1, 3, 4, 17]. 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 [28]. Cooperative concurrency is the dominant form of concurrency in distributed systems and process modeling, and is an essential element of workflow [23, 36, 29].

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 [40]. 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 [28]. 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.

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 [10]. $\mathcal{CTR}$ 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 [8]. 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.* □

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 [10].

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.[1]

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 [10, 8]. 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 [7], 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 $\odot[p(b) \otimes del.p(b)]$ 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.[2] 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 [8], $\mathcal{TD}$ accounts for many basic properties of "advanced" transaction models [24], 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

[1]Another approach would be to prove *relativized* complexity results with elementary operations modeled by a variation of oracle Turing machines [30]. Most of the results in this paper can be relativized in this way.

[2]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.

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:

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

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

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 [10, 8, 11], and are reviewed in the long version of this paper [9]. This section illustrates the procedural interpretation through a series of examples on workflow. Each example has been tested on our prototype implementation [32, 38], and performs exactly as described below.

The examples focus on so-called *production* workflow, which forms the core of a business or enterprise [26, 35]. 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 [14, 15, 39], which is engaged in several large-scale genome mapping and sequencing projects [20]. 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 [14]. 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 [15, 13, 14]. 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.

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

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.[3] 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. □

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. The rules simulate the pipelining found in many workflow systems: items enter the workflow sequentially, but are processed by the workflow concurrently. We assume that a record identifying each work item is initially stored in a database relation called $item$, which acts as an "in basket" for the workflow. The first rule recursively removes one work item after another from the in basket. After each removal, the rule spawns a workflow instance to process the item, where different instances of the workflow execute concurrently. The second rule terminates the recursion (and the simulation) when there are no work items left to process.

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

The third rule retrieves a work item, W, and then deletes it from the database. 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. □

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 [36, 29], 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

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

[3] 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.

inserts a sequence of n new work items, $w_1, w_2, ..., w_n$, one item at a time. In this way, the database 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 6.2, 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 [26]. 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 [26]. 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 [26, 15].

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.

$$
\begin{aligned}
task_i(W) \;\leftarrow\;& request_i(A) \otimes doTask_i(A, W) \otimes \\
& release(A) \otimes ins.done_i(A, W) \\
request_i(A) \;\leftarrow\;& \\
& \odot[qualified_i(A) \otimes available(A) \otimes del.available(A)] \\
release(A) \;\leftarrow\;& ins.available(A)
\end{aligned}
$$

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 [15], in which the work items are DNA samples, and the purpose of the workflow is to construct a physical genome map.[4]

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.

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

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. □

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 to describe the transactional properties of $\mathcal{TD}$ programs, to show that Transaction Datalog is an extension of classical Datalog, and 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 $\mathbf{P}$,

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

[4] 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.

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

$$\mathbf{P}, \{ab\}\,\{\,\} \models p$$

$$\mathbf{P}, \{\,\}\,\{cd\} \models q$$

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

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 [10, 8], and is reviewed in detail in the long version of this paper [9]. This semantics integrates queries, updates and concurrency in a simple logical framework.

As in the classical theory of concurrency control [7], $\mathcal{TD}$ has an interleaving semantics. Intuitively, a $\mathcal{TD}$ program consists of a number of concurrent processes, where each process generates a sequence of queries and updates. 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 (as in Example 3.4), then the write operation must come before the read operation in a legal interleaving. When an interleaved sequence is executed, it defines a transformation from databases to databases, which we describe formally by executional entailments.

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.[5] 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.

[5] The effects of *non-isolated* execution have a more complex description involving execution histories [10, 8, 9].

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

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:

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

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:

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

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 is given in the long version of this paper [9].

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 [3, 4, 16]. 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.

Due to space limitations, proofs of the theorems below are given only in the long version of this paper [9].

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 [3, 4, 17]. 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 [3, 4, 1] or they expand the database schema [16, 1]. 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 [40]. However, in CCS, the simulation relies heavily on the *restriction operator* [40, 19], 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 [6]. An example is the workflows at the Whitehead Institute/MIT Center for Genome Research [14, 39], 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 6.2 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.[6] 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 elementary operation enables $\mathcal{TD}$ to express *all* computable transactions, both monotonic and non-monotonic, as long as they are safe.[7]

[6] 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.

[7] 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 [3, 4]. Details are given in the long version of this paper [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 [3, 4, 17]. 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 [18].

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 [30], 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.*

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 **P** be a $\mathcal{TD}$ rulebase. A rule in **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 **P** exhibits sequential tail recursion if every recursive rule in **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 [15]. 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.

Theorem 5.11 *$\mathcal{TD}$ 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 5.11 does not require concurrent composition at all.

Corollary 5.12 *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. Due to space limitations, we give only a brief and informal description of the restriction here, and state the main result. A formal development is given in the long version of this paper [9]. An extended example based on cooperating workflows in a genome laboratory is given at the end of this section.

Full boundedness is based on two ideas. First, each recursive call to a predicate must remove a tuple from a base relation. For instance, in Example 3.2, each recursive call to *simulate* removes a tuple from the *item* relation. Second, tuples that are removed from a relation in this fashion must not find their way back into the relation, as in the following example:

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

Here, p moves tuples from $item_1$ to $item_2$, and q moves them back again. Thus, if p and q are executed concurrently, there is no guarantee of termination. To eliminate this possibility, we define a data flow graph that keeps track of the flow of tuples from one base relation to another at each level of recursion. If this graph is acyclic, then we say that the rulebase is fully bounded.

Theorem 6.1 *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 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 6.2 (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 $workflow_3$. However, work items are not handed directly from one workflow to another. Instead, $workflow_1$ 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 the two producer workflows, $workflow_1$ and $workflow_2$. As in Example 3.2, a record identifying each work item for $workflow_i$ is initially stored in a database relation called $item_i$, which acts as an "in basket" for the workflow. In addition, after each work item has been processed, it is inserted into the relation $basket_i$, which acts as an "out basket" for the workflow.

$$produce_i \leftarrow simulate_i \otimes ins.finished_i$$

$$simulate_i \leftarrow getItem_i(W) \otimes [simulate_i \mid (workflow_i(W) \otimes putItem_i(W))]$$

$$simulate_i \leftarrow item_i.empty$$

$$getItem_i(W) \leftarrow \odot[item_i(W) \otimes del.item_i(W)]$$

$$putItem_i(W) \leftarrow ins.basket_i(W)$$

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 item is put into the out basket. The third rule terminates the recursion (and the simulation) when there are no work items left to process.

The next set of rules simulate the execution of $workflow_3$. This workflow has two inputs, $basket_1$ and $basket_2$. The workflow selects a pair of items, one from each basket, and processes them together. This process is repeated until one of the inputs is permanently empty, at which time, the workflow stops.

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

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

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

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

$$selectItems(I, J) \leftarrow \odot[basket_1(I) \otimes basket_2(J) \otimes suitable(I, J) \otimes del.basket_1(I) \otimes del.basket_2(J)]$$

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 $basket_1$ and $basket_2$, 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 (and the simulation) when one of the inputs is permanently empty. Specifically, the third rule stops the simulation when $workflow_1$ has finished and $basket_1$ is empty. Likewise, the fourth rule stops the simulation when $workflow_2$ has finished and $basket_2$ 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:

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

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 [12, 11]. 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 [8, 10].

Workflow and Processes: The database-theory community has recently begun to study business processes and workflow management. For instance, Wodtke and Weikum have used state charts [27] to develop a formal foundation for workflow execution in a distributed environment [42]. 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 [21]. 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 [1]. 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 [28]. This feature is essential to workflow, but is lacking in traditional database languages, including all the languages described in [1].

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 [1]. 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 [18].

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.*, [33, 41, 2, 1]), 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 [2]. 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 [2], 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 [2]. 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 long version of this paper [9]. 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 [2], 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 [2].

Finally, it is worth noting that our notion of transaction expressiveness is different from that developed by Abiteboul and Vianu in [3, 4]. 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. 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. 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.

References

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

[2] 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.

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

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

[5] 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.

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

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

[8] 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.

[9] A.J. Bonner. Workflow, transactions, and datalog. In *ACM Symposium on Principles of Database Systems*, Philadelphia, PA, May/June 1999. Long version available at `http://www.cs.toronto.edu/~bonner/papers.html#transaction-logic`.

[10] 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.

[11] 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.

[12] A.J. Bonner and M. Kifer. The state of change: A survey. In [25]. 1998.

[13] 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.

[14] 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.

[15] 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`.

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

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

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

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

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

[21] 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.

[22] 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.

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

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

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

[26] 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.

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

[28] 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.

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

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

[31] 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.

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

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

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

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

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

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

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

[39] 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.

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

[41] 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.

[42] 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.