Concurrent Transaction Logic:
Semantics and Proof Theory

Anthony J. Bonner*
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4, Canada
bonner@db.toronto.edu
++1-416-978-7441 (phone)
++1-416-978-1676 (fax)

Michael Kifer†
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11790, U.S.A.
kifer@cs.sunysb.edu
++1-516-632-8459 (phone)
++1-516-632-8334 (fax)

April 20, 1995

Abstract

In an earlier work, we developed sequential Transaction Logic, which deals with state changes in logic programs and databases. It provides a framework for tasks ranging from transaction specification and execution in databases, to view updates, to triggers in active databases, to discrete system simulation, to robot planning and procedural knowledge in AI. In the present paper, we propose *Concurrent Transaction Logic* (abbr. $\mathcal{CTR}$), which extends Transaction Logic with connectives for modeling the concurrent execution of complex actions. The concurrent actions execute in an interleaved fashion and can also communicate and synchronize themselves. All this is provided in a completely logical framework, including a natural model theory and a proof theory. Moreover, the framework is flexible in that it can accommodate many different semantics for updates and for databases. For instance, not only can updates insert and delete tuples, they can also insert and delete null values, rules, or arbitrary logical formulas. Likewise, not only can databases have a classical semantics, they can also have the well-founded semantics, the stable-model semantics, etc. Finally, the proof theory for $\mathcal{CTR}$ has an efficient SLD-style proof procedure. As in the sequential version of the logic, this proof procedure not only finds proofs, it also *executes* concurrent transactions, finds their execution schedules, and updates the database. A main result is that the proof theory is sound and complete for the model theory.

Keywords: databases, updates, actions, concurrent programming, communicating processes, inference, semantics, soundness and completeness.

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

1 Introduction

This paper introduces *Concurrent Transaction Logic* (abbr. $\mathcal{CTR}$), which is an extension of Transaction Logic, described in [7, 8, 5, 6]. Transaction Logic is a logical formalism for dealing with state changes in logic programs and databases. It is suitable for a wide range of tasks, from declarative programming of database transactions and triggers, to robot simulation and planning in AI, to specification of temporal constraints on dynamic actions. It seamlessly integrates the above dynamic features with traditional deductive databases, providing a solid foundation for both [7]. Concurrent Transaction Logic builds on the previous work and extends it with constructs that allow complex actions to interleave and communicate. In this paper, we develop a model-theoretic semantics for $\mathcal{CTR}$, and we present a sound and complete proof theory for a subset of $\mathcal{CTR}$ that we call *Concurrent Horn* programs.

In [7, 6], we provide extensive comparison of Transaction Logic with a large number of proposals based on modal logic, dynamic logic, process logic, and temporal logics (*e.g.*, [21, 14, 30, 35]); proposals based on classical logic, such as the situation calculus and event calculus (*e.g.*, [22, 33, 20]); and works in databases (*e.g.*, [1, 9, 31, 10]). It is impossible in this short paper to do justice to all this body of work. In [7] we argue that although some approaches are more developed in their ability to *reason* about actions, none is as comprehensive as Transaction Logic in the *specification* and *execution* of transactions; and none of these works spans such a wide range of application domains.

Most importantly, few (if any) of the above works address the issues that are rapidly gaining momentum in all areas of Computer Science: *concurrency* and *communication*. In fact, to the best of our knowledge, $\mathcal{CTR}$ is the first comprehensive yet purely logical proposal for *programming* and *executing* concurrent and communicating database processes. Although there is a large body of work on systems for specifying and verifying concurrent systems (*e.g.*, [28, 15]), these are primarily logics or algebras for reasoning about concurrency or for specifying processes at a very high level. They are not logics suitable for actual *programming* of such processes in the declarative style of logic programming and deductive databases. Section 5 discusses these works in more detail.

Like Transaction Logic, $\mathcal{CTR}$ provides a general mechanism for combining elementary operations into complex actions. Elementary operations might include retrieving a single tuple from a relational database, inserting a tuple, or deleting a tuple. They might also include complex numerical calculations, such as fast Fourier transforms. However, the precise set of elementary operations is orthogonal to $\mathcal{CTR}$. $\mathcal{CTR}$ treats a database as a collection of abstract datatypes, each with its own special-purpose access methods.[1] These methods are combined into complex actions by $\mathcal{CTR}$ programs. This approach separates the issue of specifying elementary operations from the issue of combining them. In this way, we can study the problems of concurrent and serial execution separately, without committing to a particular theory of elementary operations. The technical device we use to achieve this is a pair of oracles: a *data oracle*, which provides primitive database queries; and a *transition oracle*, which provides primitive database updates. Elementary operations are specified outside of $\mathcal{CTR}$ via these oracles.

By factoring out the specification of elementary operations, $\mathcal{CTR}$ can accommodate many different semantics for updates and databases. For instance, not only can updates insert and delete tuples, they can also insert and delete null values, or rules, or arbitrary logical formulas. Likewise, not only can databases have a classical semantics, they can also have a well-founded semantics, a stable-model semantics, etc. A database can even consist of apparently non-logical objects, such as email queues and communication channels. We develop this latter idea into a logical facility for communication between actions.

2 Syntax

2.1 Transaction Formulas

The alphabet of a $\mathcal{CTR}$ language, $\mathcal{L}$, consists of the following symbols:

- A set of function symbols $\mathcal{F}$. Each function symbol has a non-negative number, called its *arity*, indicating the number of arguments the symbol can take. Constants are viewed as 0-ary function symbols.
- A countably-infinite set of variables $\mathcal{V}$.
- A countable set $\mathcal{P}$ of predicate symbols. Like functions, predicate symbols have arity; 0-ary predicate symbols are viewed as propositional constants.
- The logical connectives $\wedge$, $\otimes$, $|$, $\neg$, the modal operator $\odot$, and the quantifier $\forall$.

[1] This generalizes the notion of database as originally conceived in Transaction Logic, where a database was a collection of first-order formulas [6, 5].