Database Programming in Transaction Logic

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

Michael Kifer†
SUNY at Stony Brook
Department of Computer Science
Stony Brook, NY 11790, U.S.A.
kifer@cs.sunysb.edu

Mariano Consens
University of Toronto
Department of Computer Science
Toronto, Ontario M5S 1A4, Canada
consens@db.toronto.edu

Abstract

This paper presents database applications of the recently proposed *Transaction Logic*—an extension of classical predicate logic that accounts in a clean and declarative fashion for the phenomenon of state changes in logic programs and databases. It has a natural model theory and a sound and complete proof theory, but, unlike many other logics, it allows users to *program transactions*. In addition, the semantics leads naturally to features whose amalgamation in a single logic has proved elusive in the past. Finally, Transaction Logic holds promise as a logical model of hitherto non-logical phenomena, including so-called *procedural knowledge* in AI, and the *behavior* of object-oriented databases, especially methods with side effects. This paper focuses on the applications of $\mathcal{TR}$ to database systems, including transaction definition and execution, nested transactions, view updates, consistency maintenance, bulk updates, non-determinism, sampling, active databases, dynamic integrity-constraints, hypothetical reasoning, and imperative-style programming.

Published in *Proceedings of the fourth International Workshop on Database Programming Languages (DBPL)*, pages 309-337. Springer Verlag, Workshops in Computing Series. Workshop held in New York City, NY, 30 Aug–1 Sept, 1993.

This paper is available at the following URL:
ftp://db.toronto.edu/pub/bonner/papers/transaction.logic/dbpl93.ps.gz

*Work supported in part by an Operating Grant from the Natural Sciences and Engineering Research Council of Canada and by a Connaught Grant from the University of Toronto.

†Supported in part by NSF grant CCR-9102159 and a grant from New York Science and Technology Foundation. Work done during sabbatical year at the University of Toronto. Support of Computer Systems Research Institute of University of Toronto is gratefully acknowledged.

1 Introduction

Transaction Logic (abbreviated $\mathcal{TR}$) accounts in a clean, declarative fashion for the phenomenon of updating arbitrary logical theories, most notably, databases and logic programs. Unlike most logics of action, $\mathcal{TR}$ is a declarative formalism for specifying and executing procedures that update and permanently change a database, a logic program or, more generally, a logical theory. As a special case, transactions can be defined as logic programs. This is possible because, like classical logic, $\mathcal{TR}$ has a "Horn" version that has *both* a procedural and a declarative semantics, as well as an efficient SLD-style proof procedure. Since the formal aspects of $\mathcal{TR}$ can be found in [12, 10, 11], this paper focuses on the applications of $\mathcal{TR}$ to database systems.

$\mathcal{TR}$ was designed with several applications in mind, especially in databases, logic programming, and AI. It was therefore developed as a general logic, so that it could solve a wide range of update-related problems. Individual applications can be carved out of different fragments of the logic. These applications, both practical and theoretical, are discussed in great detail in [10]. For instance, in logic programming, $\mathcal{TR}$ leads to a clean, logical treatment of the *assert* and *retract* operators in Prolog, which effectively extends the theory of logic programming to include updates as well as queries. In object-oriented databases, $\mathcal{TR}$ can be combined with object-oriented logics, such as F-logic [23], to provide a logical account of *methods*—procedures hidden inside objects that manipulate these objects' internal states. Thus, while F-logic covers the structural aspect of object-oriented databases, its combination with $\mathcal{TR}$ would account for the behavioral aspect as well. In AI, $\mathcal{TR}$ suggests a logical account of planning. STRIPS-like actions,[1] for instance, and many aspects of hierarchical and non-linear planning are easily expressed in $\mathcal{TR}$. In spite of the previous efforts to give these phenomena declarative semantics, until now there has been no unifying *logical* framework to account for all of them.

On the surface, there would seem to be many other candidates for a logic of transactions, since many logics reason about updates or about the related phenomena of time and action. However, despite a plethora of action logics, researchers continue to complain that there is no clear declarative semantics for updates, whether in databases or in logic programming [7, 5, 28]. In fact—in stark contrast to classical logic—no action logic has ever become a core of databases or logic-programming, in theory or in practice. There appear to be a few simple reasons for this unsuitability of existing action logics. These reasons are discussed at length in [10], and we discuss some of them briefly here.

First, most logics of time or action are *hypothetical*. For instance, some systems can infer that if action A precedes B, and B precedes C, then A must precede C. Others can infer that if a student took history 400, then he could graduate. Such systems were intended to be observers of action, not participants. They are therefore useful for reasoning about alternatives, or for analyzing programs and plans; but they are not very useful for defining procedures that actually *accomplish* state changes being reasoned about. In $\mathcal{TR}$, actions can be carried out hypothetically or they can be executed and have a permanent effect on the database, depending on one's desire. Furthermore, the proof theory of $\mathcal{TR}$ is not only a verifier of truth, but also an *executor* of transactions.

[1] STRIPS was an early AI planning system that simulated the actions of a robot arm.

Second, many logics make a clear distinction between queries and updates. However, this distinction is blurred in object-oriented systems, where both queries and updates are special cases of a single idea: method invocation. In such systems, an update can be thought of as a query with side effects. We would like to model this behavior and thereby provide a logical foundation for object-oriented databases. $\mathcal{TR}$ achieves this by allowing every logical formula to have not only a truth value, but also a "side effect" on the database. In this way, one can account for the *behavior* of object-oriented databases—something that most formalisms do not do. By integrating $\mathcal{TR}$ with F-logic [23], the structural aspect of object-oriented systems can be accounted for as well.

The system that comes closest in *spirit* to $\mathcal{TR}$ is Prolog. Unfortunately, updates in Prolog are non-logical operations and, as a result, state-changing procedures are often the most awkward of Prolog programs, and the most difficult to understand, debug, and maintain. $\mathcal{TR}$ provides a general solution to the aforementioned limitations, both of Prolog and of action logics.

2 Overview of Transaction Logic

$\mathcal{TR}$ is an extension of first-order logic, both syntactically and semantically. It also has a natural model theory and a sound-and-complete proof theory. This section gives an overview of the syntax and the model theory. A complete development of $\mathcal{TR}$, including proof theory, can be found in [10] (and to some extent in [12]).

Like classical logic, $\mathcal{TR}$ has a "Horn" version that is of particular interest for deductive databases. In Horn $\mathcal{TR}$, a transaction is defined by Datalog-style rules in which the premise specifies a *sequence* of queries and updates. Horn $\mathcal{TR}$ is thus a logical language for programming database transactions, just as Datalog is a logical language for programming queries. Furthermore, Horn $\mathcal{TR}$ has an efficient SLD-style proof procedure and also a dual, bottom-up procedure [12, 10]. These proof procedures answer queries, execute transactions, *and* update the database. Because of its importance, much of this paper focuses on applications of Horn $\mathcal{TR}$, but first we describe full $\mathcal{TR}$, without the Horn restriction.

2.1 Syntax

The syntax of $\mathcal{TR}$ distinguishes two kinds of formulas: *transaction formulas* and *elementary transitions*. The former define composite transactions, and the latter define elementary updates.

Transaction formulas are used to define transactions and formulate queries. Transaction formulas extend first-order formulas with a new connective, $\otimes$, called *serial conjunction*. Formally, transaction formulas are defined recursively as follows. An *atomic* transaction formula is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol, and $t_1, \ldots, t_n$ are terms (as in classical predicate calculus). If ϕ and ψ are transaction formulas, then so are $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \otimes \psi$, $\neg\phi$, $(\forall X)\phi$, and $(\exists X)\phi$, where X is a variable. Thus, the expression $a(X) \vee \neg[b(X) \otimes c(X,Y)]$ is a transaction formula. Informally, $\psi \otimes \phi$ says, "Do ψ and then do ϕ." A dual connective, *serial disjunction*, is also useful (Section 3.8): $\psi \oplus \phi$ is equivalent to

$\neg(\neg\phi \otimes \neg\psi)$.

Serial conjunction provides a basic way to *sequence* transactions, where $\phi \otimes \psi$ means "do ϕ then do ψ." In contrast, classical conjunction, "$\wedge$", constrains the non-determinism of a transaction. For instance, $\phi \wedge \psi$ means, "do ϕ in a way compatible with doing ψ." This use of "$\wedge$" is further discussed in Section 3.8. Apart from this, "$\wedge$" also has the traditional role of forming logic programs: in $\mathcal{TR}$, as in classical logic, any finite set of rules is equivalent to a conjunction of all the rules in the set. In $\mathcal{TR}$, such a set of transaction formulas is called a *transaction base.*

A transaction base defines complex formulas in terms of simpler ones. However, we also need a way to specify *elementary changes* to a database. One way to define such transitions is to build them into the semantics as in [25, 27, 16, 2]. A problem with this approach is that adding new kinds of elementary transitions leads to a redefinition of the very notion of a model and thus to an overhaul of the entire proof theory. This is a serious drawback since there appears to be no small, single set of elementary transitions that is best for all purposes [10]. Indeed, Sections 3.3 and 3.4 introduce two new kinds of elementary update. Thus, rather than committing $\mathcal{TR}$ to a fixed set of elementary transitions, we have chosen to treat the elementary transitions as a *parameter* of $\mathcal{TR}$. Each set of elementary transitions thus gives rise to a different version of the logic. To achieve this, elementary transitions are defined by logical axioms.

Elementary transitions are formulas of the form $\langle\phi, \psi\rangle u$, where ϕ, ψ are (sets of) closed first-order formulas and u is an atomic formula, called the *name* of the transition. Intuitively, this formula says that u is an update that transforms database ϕ into database ψ. For instance, if the atoms $ins{:}q(t)$ and $del{:}q(t)$ stand for the insertion and deletion of atom $q(t)$, then they would be defined by an enumerable set of elementary transitions consisting of the following formulas:

$$\langle \mathbf{D}, \mathbf{D} + \{q(t)\}\rangle\ ins{:}q(t) \qquad\qquad \langle \mathbf{D}, \mathbf{D} - \{q(t)\}\rangle\ del{:}q(t)$$

for every relational database $\mathbf{D}$.[2] Enumerable sets of elementary transitions are called *transition bases.* In practice, these formulas would not be materialized all at once, but would be generated on demand by an algorithm. The reader is referred to [10] for a more detailed discussion of transition bases.

As seen from the above syntax, there is no strict distinction in $\mathcal{TR}$ between predicates that query the database and predicates that update it. As in classical logic, every predicate has a truth value, but in addition, every predicate may have a side effect by changing the state of the database. This uniformity of representation is important for modeling *methods* in object-oriented databases, where one generally does not distinguish between information-retrieving and state-changing methods. Nevertheless, if desired, $\mathcal{TR}$ can make such a distinction by using different sorts of predicates, one for updates and one for queries. For instance, it may be a good programming practice to reserve a special set of predicates for certain basic updates. This paper uses just such a convention: for each predicate symbol p, we use another predicate symbol, *ins:p*, to represent insertions of tuples into p. Likewise, we represent deletions from p

[2] For relational databases, the operators $+$ and $-$ can be thought of as union and set-difference. However, if $\mathbf{D}$ is a general first-order formula, then defining insertion and deletion is more involved [22].

by the predicate *del:p*. Thus the formula $ins{:}p(a) \otimes ins{:}p(b) \otimes ins{:}p(c)$ represents an updating transaction that inserts $p(a)$ into the database, then $p(b)$, and then $p(c)$.

2.2 An Example

This section gives a simple example of a transaction base. The body of each rule is a sequence of atomic formulas, some of which are queries and some of which are updates. The example shows how updates can be combined with queries to define complex transactions. It also illustrates the use of transaction subroutines (or nested transactions), and shows how $\mathcal{TR}$ improves upon Prolog's update operators.

Example 2.1 (Financial Transactions) Suppose the balance of a bank account is given by the relation $balance(Acct, Amt)$. To modify this relation, we are provided with a pair of elementary update operations: $del{:}balance(Acct, Amt)$ to delete a tuple from the relation; and $ins{:}balance(Acct, Amt)$, which inserts a tuple into the relation. Using these two updates, we define four transactions: $change{:}balance(Acct, Bal1, Bal2)$ to change the balance of an account from one amount to another; $withdraw(Amt, Acct)$ to withdraw an amount from an account; $deposit(Amt, Acct)$ to deposit an amount into an account; and $transfer(Amt, Acct1, Acct2)$ to transfer an amount from one account to another. These transactions are defined by the following four rules, which form a transaction base:

$$
\begin{array}{l}
transfer(Amt, Acct1, Acct2) \leftarrow withdraw(Amt, Acct1) \otimes deposit(Amt, Acct2) \\
withdraw(Amt, Acct) \leftarrow balance(Acct, Bal) \otimes Bal \geq Amt \\
\qquad\qquad\qquad\qquad\qquad\qquad \otimes change{:}balance(Acct, Bal, Bal - Amt) \\
deposit(Amt, Acct) \leftarrow balance(Acct, Bal) \\
\qquad\qquad\qquad\qquad\qquad\qquad \otimes change{:}balance(Acct, Bal, Bal + Amt) \\
change{:}balance(Acct, Bal1, Bal2) \leftarrow del{:}balance(Acct, Bal1) \\
\qquad\qquad\qquad\qquad\qquad\qquad \otimes ins{:}balance(Acct, Bal2)
\end{array}
$$

In each rule, the premises are evaluated from left to right—an evaluation order imposed by the serial conjunction, $\otimes$. For instance, the first rule says: to transfer an amount from $Acct1$ to $Acct2$, first withdraw the amount from $Acct1$ and, if the withdrawal succeeds, deposit the amount in $Acct2$. Likewise, the second rule is interpreted thus: to withdraw an amount, Amt, from an account, $Acct$, first retrieve the balance of the account; then check that the account will not be overdrawn by the transaction; if all is well, change the balance from Bal to $Bal - Amt$. Notice that the atom $balance(Acct, Bal)$ is a query that retrieves the balance of the specified account and $Bal \geq Amt$ is a test. All other atoms in this example are updates. The fourth rule changes the balance of an account by deleting the old balance and then inserting the new balance. Unlike the other rules, this rule is defined in terms of built-in, elementary updates, *del:balance* and *ins:balance*. □

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

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

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

Getting around this problem takes much out of the simplicity of Prolog programming. In fact, the non-logical behavior of Prolog updates is notorious for making Prolog programs cumbersome and heavily dependent on Prolog's backtracking strategy. $\mathcal{TR}$ fixes this problem by providing a simple logical semantics for database updates.

2.3 Model Theory

This section discusses the model theory of $\mathcal{TR}$. For easy reference, some details are given in Appendix A; the reader is referred to [10] for a full treatment.

Just as the syntax of $\mathcal{TR}$ is based on two basic ideas—serial conjunction and elementary transitions—semantics is also based on a few fundamental principles:

- *Transaction Execution Paths*: A transaction causes a sequence of database state changes;
- *Database States*: A database state is a *set* of (classical) first-order semantic structures;
- *Executional Entailment*: Transaction execution corresponds to truth over a sequence of states.

Transaction Execution Paths: When the user executes a transaction, the database may change, going from some initial state to some final state. In doing so, the database may pass through any number of intermediate states. For example, execution of the transaction $\mathit{ins{:}a} \otimes \mathit{ins{:}b} \otimes \mathit{ins{:}c}$ takes the database from an initial state, $\mathbf{D}$, through the intermediate states $\mathbf{D} + \{a\}$ and $\mathbf{D} + \{a, b\}$, to the final state $\mathbf{D} + \{a, b, c\}$. This idea of a sequence of states is central to our semantics of transactions. It also allows us to model a wide range of constraints. For example, we may require that every intermediate state satisfy some condition, or we may forbid certain sequences of states.

To model transactions, we start with a modal-like semantics, where each state represents a database, and each elementary update causes a transition from one state to another, thereby changing the database. At this point, however, modal logic and Transaction Logic begin to part company. The first major difference is that truth in $\mathcal{TR}$ structures does not hinge on a set of arcs between states. Instead, we focus on *paths*, that is, on sequences of states. (This focus on paths is related to the version of Process Logic in [20], but the two logics are fundamentally different [10].) Because of the emphasis on paths, we refer to semantic structures in $\mathcal{TR}$ as *path structures*. Second, truth in path structures is defined on paths, not states. For example, we would say that the path $\langle \mathbf{D},\ \mathbf{D}+\{a\},\ \mathbf{D}+\{a,b\} \rangle$ satisfies the formula $ins{:}a \otimes ins{:}b$, since it represents an insertion of a followed by an insertion of b. A path of length 1 corresponds to a single database state. In this way, one model-theoretic device, paths, accounts for databases, updates, queries and more general transactions.

Other logical connectives are also interpreted on paths, *i.e.*, in terms of action. For instance, $\phi \vee \psi$ is true on a path iff ϕ is true or ψ is true. This gives rise to non-deterministic actions, since intuitively, $\phi \vee \psi$ means, "Do ϕ or do ϕ." Section 3.4 illustrates this idea. Likewise, $\phi \wedge \psi$ is true on a path iff ϕ and ψ are both true. This provides a way of constraining non-deterministic actions. For instance, $\phi \wedge \neg\psi$ intuitively means, "Do ϕ but without doing ψ in the process." Section 3.8 illustrates this idea. Finally, note that $\otimes$ and $\wedge$ are identical on paths of length 1. Thus, on states, $\mathcal{TR}$ reduces to classical logic.

Database States: Another difference between modal logic and Transaction Logic is in the nature of states. In modal logic, a state is basically a first-order semantic structure, since each state specifies the truth of a set of ground atomic formulas. Such structures are adequate for representing relational databases, but not for representing more general theories, like indefinite databases or general logic programs. We therefore take a more general approach. Since a database is a first-order formula, which has a *set* of first-order models, we define a state to be a *set* of first-order semantic structures. Each database thus corresponds to a particular state—the state consisting of all the models of the database.

This approach to states provides a lot of flexibility when defining elementary updates. Such flexibility is needed since, for general databases, the semantics of elementary updates is not obvious, not even for relatively simple updates like insert and delete. For example, what does it mean to insert an atom b into a database that entails $\neg b$, especially if $\neg b$ itself is not explicitly present in the database? There is no simple answer to this question, and many solutions have been proposed (see [22] for a comprehensive discussion). For these reasons, we take a general approach to elementary updates. For us, an elementary update is a mapping that takes each database $\mathbf{D}_1$ to some other database $\mathbf{D}_2$, where a database is any first-order formula. More generally, an elementary update may be non-deterministic, so it is not just a mapping, but a *binary relation* on databases.

3 Database Applications

A wide variety of interesting and useful formulas can be constructed in $\mathcal{TR}$, formulas that capture many of the novel and important features of database and knowledge-base systems. These features include transaction definition and execution, ad hoc queries, view updates, consistency maintenance, bulk updates, non-determinism, sampling, dynamic integrity-constraints, invented values, and more. This section describes some of these applications; more can be found in [10]. We shall also see that the semantics of $\mathcal{TR}$ allows the easy introduction of a modal necessity operator, $\Box$, which captures a whole new range of applications. These applications include hypothetical reasoning, imperative programming constructs, active databases, software verification, and more. $\mathcal{TR}$ thus provides a wide range of features whose amalgamation in a single declarative formalism has proved elusive in the past. Furthermore, these features all follow naturally from $\mathcal{TR}$'s path-based semantics.

3.1 Consistency Maintenance

Updating one relation often entails the need for additional updates to other relations in order to maintain the semantic consistency of the database. In such cases, updates to a relation can be done through special procedures that handle the details of consistency maintenance. Such procedures are easily defined in $\mathcal{TR}$.

For example, suppose a university has a database of students, courses, and professors. This database includes the following four base relations:

- *takes(Stud,Crs,Sec)*, which records the students enrolled in each section of each course.
- *enrolled(Crs,N)*, which records the total number of students, N, enrolled in a course.
- *instructs(Prof,Crs,Sec)*, which records the professors who teach each section of each course.
- *load(Prof,N)*, which records each professor's course load, N, *i.e.*, the total number of classes that he teaches.

Per the convention adopted in this paper, we assume that for each base relation, p, the transition base underlying the database system defines two elementary update predicates, *ins:p* and *del:p*, for inserting and deleting tuples from relation p.

Using these elementary updates, we define two update-procedures by which students drop courses and professors are relieved from teaching sections of a course. These procedures ensure database consistency by decrementing the enrollment total when a student drops a course, and by decrementing a professor's course load when he is relieved from teaching a course.

$$
\begin{array}{rcl}
drop(Stud, Crs, Sec) & \leftarrow & takes(Stud, Crs, Sec) \otimes del{:}takes(Stud, Crs, Sec) \\
 & & \quad \otimes\ decr\_enrolled(Crs) \\
relieve(Prof, Crs, Sec) & \leftarrow & instructs(Prof, Crs, Sec) \otimes \\
 & & \quad del{:}instructs(Prof, Crs, Sec) \otimes decr\_load(Prof) \\
decr\_enrolled(Crs) & \leftarrow & enrolled(Crs, N) \otimes del{:}enrolled(Crs, N) \otimes \\
 & & \quad ins{:}enrolled(Crs, N-1) \\
decr\_load(Prof) & \leftarrow & load(Prof, N) \otimes del{:}load(Prof, N) \otimes \\
 & & \quad ins{:}load(Prof, N-1)
\end{array}
$$

The last two rules define procedures for decrementing the enrollment of a course and the teaching load of a professor, respectively.

3.2 View Updates

Updating a view is often an ill-defined or non-deterministic process, since changes to a view may not uniquely determine the corresponding changes to the underlying stored database. To illustrate the problems and some solutions, consider the university database of Section 3.1 to which we add the following view definition that indicates which professors teach which courses to which students:

$$\begin{aligned} \mathit{teaches}(\mathit{Prof}, \mathit{Stud}, \mathit{Crs}) \ \leftarrow \ & \mathit{takes}(\mathit{Stud}, \mathit{Crs}, \mathit{Sec}) \\ & \otimes \mathit{instructs}(\mathit{Prof}, \mathit{Crs}, \mathit{Sec}) \end{aligned}$$

Since *teaches* is not a base predicate, the transition base does not provide update-procedures for it. The problem is that such updates are underspecified, since an update to *teaches* must be carried out in terms of updates to the base predicates *takes* and *instructs*. For instance, a deletion from *teaches* requires either a deletion from *takes* or a deletion from *instructs*. Since there are two choices, this view update is non-deterministic.

$\mathcal{TR}$ offers two solutions to this problem. The first one is to define a distinct procedure for each allowed way of deleting a tuple from a view. The second solution is based on defining a non-deterministic transaction for removing tuples from a view.

To illustrate the first approach, we define a procedure called *rem_student* that allows a user of the view to remove a student from a course. Likewise, we define a procedure called *rem_prof* that allows a user to remove a professor from a course. These two procedures are defined as follows:

$$\begin{aligned} \mathit{rem\_student}(\mathit{Prof}, \mathit{Stud}, \mathit{Crs}) \ \leftarrow \ & \mathit{takes}(\mathit{Stud}, \mathit{Crs}, \mathit{Sec}) \\ & \otimes \mathit{drop}(\mathit{Stud}, \mathit{Crs}, \mathit{Sec}) \\ \mathit{rem\_prof}(\mathit{Prof}, \mathit{Stud}, \mathit{Crs}) \ \leftarrow \ & \mathit{takes}(\mathit{Stud}, \mathit{Crs}, \mathit{Sec}) \\ & \otimes \mathit{relieve}(\mathit{Prof}, \mathit{Crs}, \mathit{Sec}) \end{aligned}$$

In this way, a user can do view deletions without knowing what section of a course a student takes or a prof teaches, and without being given direct access to the transactions *drop* and *relieve*. This approach to view updates is similar to that advocated for object-oriented databases, in which a different update method is programmed for each allowed view update [1, 6].

The second approach to the above problem is to define a non-deterministic update-procedure, *rem_teaches*, by combining the above definitions of *rem_student* and *rem_prof*:

$$\begin{aligned} \mathit{rem\_teaches}(\mathit{Prof}, \mathit{Stud}, \mathit{Crs}) \ \leftarrow \ & \mathit{takes}(\mathit{Stud}, \mathit{Crs}, \mathit{Sec}) \\ & \otimes \mathit{drop}(\mathit{Stud}, \mathit{Crs}, \mathit{Sec}) \\ \mathit{rem\_teaches}(\mathit{Prof}, \mathit{Stud}, \mathit{Crs}) \ \leftarrow \ & \mathit{takes}(\mathit{Stud}, \mathit{Crs}, \mathit{Sec}) \\ & \otimes \mathit{relieve}(\mathit{Prof}, \mathit{Crs}, \mathit{Sec}) \end{aligned}$$

To delete the fact that a professor teaches a course to a certain student, the system can perform one of two actions: (i) it can drop the student from the course, or (ii) it can relieve the professor from the course. This choice is non-deterministic and is made by the system at run time.

Of course, a user will not usually want to leave such choices entirely to the database system. In such cases, the user can constrain the system's choice, to ensure, for instance, that a deletion from the *teaches* relation does not relieve a professor from a course. To do this, he could specify the following transaction:

$$?- \ load(alberto, N) \ \otimes \ rem\_teaches(alberto, mariano, cs100) \\ \otimes \ load(alberto, N)$$

This transaction removes $teaches(alberto, mariano, cs100)$ from the view, but only if Alberto's course-load remains the same after the update. Thus, the path in which Mariano drops *cs100* will be chosen. Alternatively, the user might want to ensure that the transaction does *not* drop Mariano from the course. In this case, he would write:

$$?- \ enrollment(cs100, N) \ \otimes \ rem\_teaches(alberto, mariano, cs100) \\ \otimes \ enrollment(cs100, N)$$

This transaction succeeds only if the enrollment in the course remains the same after the transaction execution. Thus, the path in which Alberto is relieved from teaching $cs100$ will be chosen. By such means, we can constrain the way in which view updates are carried out. More generally, we can constrain the way in which any transaction is carried out, and without having to reprogram the transaction. Section 3.8 considers more sophisticated kinds of constraints.

3.3 Bulk Updates

The ability to perform bulk updates is one of the cornerstones of database languages. It is routine in such database *lingua franca* as SQL or QUEL. For example, inserting a set of tuples into a relation is a basic SQL operation, as is deleting a set of tuples from a relation. Yet, bulk updates like these are conspicuously absent from most logic-based proposals for updating logic programs. The few exceptions are [13, 16, 27, 2], which are discussed in Section 4. The unusual difficulty with this kind of update seems to arise because most logical formulations of updates are based on the insertion and deletion of *single* tuples. This is not how SQL works, however. SQL first computes a query and then inserts the resulting *set* of tuples into a relation. Deletion is handled in a similar fashion.

To capture such behavior, it appears that we need an elementary state transition that accomplishes bulk updates at the lowest level. In this section, we consider *relational assignment*, which copies the contents of one relation into another relation. Just as variable assignment is a basic operation of procedural programming languages, relational assignment can be used as a basic operation of procedural database languages [14, 15]. The rest of this section shows how to express and use relational assignment in $\mathcal{TR}$.

Unlike [25, 27, 16], specific elementary transitions are not built into the semantics of $\mathcal{TR}$. Relational assignment can therefore be added to $\mathcal{TR}$ by simply adding appropriate formulas to the transition base. To see how, consider a $\mathcal{TR}$ language, $\mathcal{L}$. For every pair of predicate symbols, r and q, of the same arity, let $\mathcal{L}$ contain a propositional constant, denoted $[r := q]$. For simplicity, we restrict our attention to deductive databases in which r is an extensional (*i.e.*, non-derived) predicate. If $\mathbf{D}$ is such a database, then let $\mathbf{D}'$ be the database derived from $\mathbf{D}$ by deleting all r-facts and replacing them by the set $\{r(t_1, \ldots, t_n) \mid \mathbf{D} \models q(t_1, \ldots, t_n)\}$ (*i.e.*, first delete the contents of relation r, and then copy the contents of relation q into r.)

Finally, let the transition base, $\mathcal{B}$, contain the formula $\langle \mathbf{D}, \mathbf{D}' \rangle [r := q]$ for every such pair of databases. Thus, the transition base contains the following formulas (among many others):

$$\langle \{q(a), q(b)\}, \quad \{q(a), q(b), r(a), r(b)\}\rangle \ [r := q]$$
$$\langle \{q(a), q(b), r(c)\}, \quad \{q(a), q(b), r(a), r(b)\}\rangle \ [r := q]$$
$$\langle \{q(a), q(b), r(c), r(d)\}, \quad \{q(a), q(b), r(a), r(b)\}\rangle \ [r := q]$$

Note that, by definition, the extent of r after executing the elementary state transition $[r := q]$ is determined entirely by the current database state. This has the following important implication: if q is defined by a set of rules where some of the rules are in the database and some are in the transaction base, then only the tuples contributed by the rules in the database will be assigned to r.

Having defined relational assignment, we can easily define bulk inserts and deletes. Suppose we wanted to add to r all tuples satisfying some condition ϕ and delete from s all tuples satisfying ψ. To do so, for each of these operations, we first define two derived relations, $q1$ for the insertion into r, and $q2$ for the deletion from s, as follows:

$$\begin{aligned} q1(X) &\longleftarrow r(X) \\ q1(X) &\longleftarrow \phi(X) \\ q2(X) &\longleftarrow s(X) \wedge \neg\psi(X) \end{aligned} \tag{1}$$

Note that the extension of $q1$ is $r \cup \phi$, and the extension of $q2$ is $s - \psi$. To actually perform the updates, we use the elementary transitions $[r := q1]$ and $[s := q2]$, which effectively insert the tuples satisfying ϕ into r, and delete the tuples satisfying ψ from s. In this way, relational assignment captures the update behavior of SQL, including the use of existential subqueries to perform bulk deletion. It should be clear from an earlier remark that the above rules must all be in $\mathbf{D}$ for the relational assignment to work as intended.

More-complex transactions are also easy to express. For example, consider the transaction, "*Raise the salary of all managers by 7%, and then retrieve all employees whose salary is greater than 100K.*" This transaction can be expressed as follows:

$$\begin{aligned} &empl2(E, Sal * 1.07, mngr) \longleftarrow empl(E, Sal, mngr) \\ &empl2(E, Sal, Rank) \longleftarrow empl(E, Sal, Rank) \wedge Rank \neq mngr \\ &result(E) \longleftarrow [empl := empl2] \otimes query(E) \\ &query(E) \longleftarrow empl(E, Sal, Rank) \otimes Sal > 100K \\ &query(\texttt{null}) \longleftarrow \end{aligned} \tag{2}$$

The new contents of the employee relation is computed by the first two rules and is held in the relation *empl*2. As explained above, both these rules must be in the database, $\mathbf{D}$, in order for the assignment $[empl := empl2]$ in the third rule to work as intended. The query $?\!-\ result(E)$ changes the database state *and* returns all suitable employees as the answer. If no employee earns in excess of 100K after the update, the update is performed anyway, but the only answer to be returned is some specifically designated value, `null`. Observe also that this transaction simultaneously involves deletion of some tuples with old salaries *and* insertion of tuples with new salaries. Of course, this combined transaction could have been expressed following the methodology for deletions and insertions described earlier, in (1). However, this would have required two relational assignments instead of one. In the above example, we defined the temporary relation *empl2* in such a way that only one relational assignment is needed.

We should also note that when the auxiliary predicates (such as $q1$, $q2$, *empl2* above) are non-recursive, it is possible to do away with these predicates and their defining rules. To this end, we can define the following, more general, form of relational assignment: $[q := (\bar{X}).\phi]$. Here ϕ is a first-order formula all of whose predicates are in **D** and $(\bar{X})$ is a list of all free variables in ϕ (with possible repetitions). We assume that the length of $\bar{X}$ equals the arity of q. This elementary state transition has the effect of assigning q the relation $\{\bar{x} \mid \phi[\bar{x}] \text{ is true}\}$ — the set of all tuples that when substituted for $\bar{X}$ makes ϕ true.[3] We can now rewrite rulebase (2) as follows:

$$
\begin{array}{l}
result(E) \longleftarrow [empl := (E, Sal, Rank).\phi] \otimes query(E) \\
query(E) \longleftarrow empl(E, Sal, Rank) \otimes Sal > 100K \\
query(\texttt{null}) \longleftarrow
\end{array}
$$

where ϕ is the following first-order formula:

$$
\begin{array}{l}
\exists S[empl(E, S, mngr) \ \wedge \ Sal = 1.07 * S] \\
\vee[empl(E, Sal, Rank) \ \wedge \ Rank \neq mngr]
\end{array}
$$

These generalized bulk updates have all the power of bulk updates in SQL, including subqueries. This is because a generalized bulk update computes an arbitrary first-order query, ϕ, and assigns its output to a base relation, q. As a special case, a bulk update can change the value of q to $q \cup \phi$, thereby expressing arbitrary SQL insertions. Likewise, a bulk update can change the value of q to $q - \phi$, thereby expressing arbitrary SQL deletions.

Finally, it is worth noting that the use of generalized bulk updates in $\mathcal{TR}$ closely parallels the embedding of SQL in procedural programming languages. In both cases, bulk updates are elementary operations invoked from a host language. And in both cases, these bulk updates can have free variables (parameters) that are bound at run time. Furthermore, in an update like $[q := (\bar{X})\phi]$, the base relation q can play the role of a *cursor*. Using the methods introduced in Section 3.6, one can iterate over the relation q one tuple at a time, just like an SQL cursor. $\mathcal{TR}$ can thus be seen as a formal basis for embedded SQL.

3.4 Non-Deterministic Sampling

In [24], Krishnamurthy and Naqvi proposed the so-called *choice*-operator. They argued that non-deterministic choice is needed to write queries such as, "Produce a sample of one employee from each department." The idea was to introduce a special construct, $choice((\bar{X}), (\bar{Y}))$, that selects those instantiations of the variables $\bar{X}$ and $\bar{Y}$ that satisfy the functional dependency (abbr., FD) $\bar{X} \rightarrow \bar{Y}$.

[3] If ϕ is a disjunction of existentially quantified conjuncts of positive literals then "truth" can be taken to mean truth in all models of ϕ. Otherwise, if literals can be negative, truth should be viewed with respect to a perfect model of ϕ. See [10] for more details.

In $\mathcal{TR}$, the *choice*-operator can be represented as another kind of elementary bulk update, one closely related to the relational assignment operator of the previous section. For example, consider the following query: *For each department with over 100 employees, choose an employee earning less than \$20K.* We can represent this query in $\mathcal{TR}$ using a pair of rules, as follows:

$$\begin{aligned} eligible(D,E) &\longleftarrow dept(E,D) \ \wedge \ size(D,N) \ \wedge \ N > 100 \\ &\qquad\qquad\qquad\qquad \wedge \ salary(E,S) \ \wedge \ S < 20K \\ answer(D,E) &\longleftarrow [sample \overset{1\rightarrow 2}{:=} eligible] \otimes sample(D,E) \end{aligned} \tag{3}$$

The first rule produces a set of candidate answers, by selecting *all* eligible employees from the appropriate departments. The second rule then samples the set of candidates, selecting *one* eligible employee per department. The heart of this rule is a new type of elementary update, $[sample \overset{1\rightarrow 2}{:=} eligible]$, that sets the extent of relation *sample* to be a subset of relation *eligible*. Any subset will do as long as it has the following two properties:

- it satisfies the specified FD, $1 \rightarrow 2$; and
- $sample[1] = eligible[1]$, *i.e.*, the projections of *sample* and *eligible* on the first attribute are equal (and thus every department is represented in the sample).

In general, we introduce an elementary update, called *sampling assignment*, denoted $[p \overset{fd}{:=} q]$, where fd is an FD, and p and q are predicate symbols of the same arity (where p is an extensional predicate). Given a database, $\mathbf{D}$, the assignment updates relation p non-deterministically. In particular, it sets the extent of p to be some subset, rel, of the relation $\{\, \bar{x} \mid \mathbf{D} \models q(\bar{x}) \,\}$ that satisfies the following two properties:

1. rel satisfies the functional dependency fd; and
2. rel is a *maximal* subrelation of $\{\bar{x} \mid \mathbf{D} \models q(\bar{x})\}$ having property 1.

As with bulk updates, sampling assignments are defined in the transition base. For instance, to define $[p \overset{1\rightarrow 2}{:=} q]$, we add the following entries to the transition base (among many others):

$$\langle \{q(a,b), q(a,c), q(a,d)\}, \quad \{q(a,b), q(a,c), q(a,d), p(a,b)\}\rangle \ [p \overset{1\rightarrow 2}{:=} q]$$

$$\langle \{q(a,b), q(a,c), q(a,d)\}, \quad \{q(a,b), q(a,c), q(a,d), p(a,c)\}\rangle \ [p \overset{1\rightarrow 2}{:=} q]$$

$$\langle \{q(a,b), q(a,c), q(a,d)\}, \quad \{q(a,b), q(a,c), q(a,d), p(a,d)\}\rangle \ [p \overset{1\rightarrow 2}{:=} q]$$

In practice, this transition base would probably be enumerated by a special procedure written in a language like C. The important point here is that the semantics of $\mathcal{TR}$ allows such procedures. Thus we do not have to explicitly construct a choice operator, unlike in [24, 30]. Instead, it falls out naturally from the semantics of $\mathcal{TR}$ as a special case.

3.5 Hypothetical Reasoning

Hypothetical queries play an important role in reasoning about knowledge [9]. Because of such queries, it is often necessary to perform hypothetical updates as well as actual ones. For instance, a game-playing program may reason as follows: After performing some given series of actions, α, does the opponent's situation improve? Observe that the actions mentioned in this query are purely hypothetical and are *not* committed. If the answer to the query is "no," then the program would perform action α, at which point the action *is* committed. Otherwise, the program would do further depth analysis and perform the most favourable move that it finds. By distinguishing between real and hypothetical actions, this program combines reasoning about action (planning, exploration of alternatives, etc) with actual execution of actions (committing itself to a particular course of action). $\mathcal{TR}$ is the only logic we are aware of that can do *both* these things.

To represent hypothetical actions, we extend the syntax of $\mathcal{TR}$. Formally, a *hypothetical formula* is an expression of the form $\Diamond\phi$ or $\Box\phi$, where ϕ is a transaction formula or a hypothetical transaction formula. Hypothetical operators can thus be nested. In modal terms, $\Diamond\,\phi$ means that the execution of ϕ is *possible* starting at the present state, and $\Box\,\phi$ means that the execution of ϕ is *necessary* at the present state. Necessity means that ϕ is executable along *every* path leaving the current state, $\mathbf{D}$. Likewise, possibility means that ϕ is executable along *some* path leaving the current state. Hypotheticals hold immediately, *i.e.*, over paths of length 1, and so they do not cause any real state transitions. The formal meaning of hypothetical formulas is given in the appendix. As shown in [10], these formulas cannot be be expressed in the version of $\mathcal{TR}$ presented so far. Thus they strictly increase the power of the language.

The next two sections explore some non-trivial application of hypothetical operators. Other applications of hypotheticals as well as a sound-and-complete proof theory for them are developed in [10].

3.6 Imperative Programming Constructs

Perhaps, one of the most interesting bonuses provided by the hypothetical operators in $\mathcal{TR}$ is the ability to express standard imperative constructs, such as **if-then-else** and **while-do** in a simple, declarative way. For instance, the following rules express an **if-then-else** statement:

$$\begin{array}{l} if\_a\_b\_c \longleftarrow (\Diamond a)\otimes b \\ if\_a\_b\_c \longleftarrow (\Box\neg a)\otimes c \end{array} \tag{4}$$

Intuitively, the query $?-\ if\_a\_b\_c$ says, "if it is possible to do a, then do b, else do c." We shall therefore write **if** $\Diamond a$ **then** b **else** c **fi** as an abbreviation for the proposition $if\_a\_b\_c$, provided that $if\_a\_b\_c$ does not occur in the head of any other rule. (Alternatively, it could abbreviate the formula $(\Diamond a \otimes b) \vee (\Box\neg a \otimes c)$, which combines the bodies of the two rules above.) Because a can be an action that changes the state of the database, the use of the hypotheticals is crucial to the proper formulation of this imperative statement. Furthermore, the negation in "$\Box\neg a$" is of the negation-by-failure variety. In [10], we present the perfect-model semantics for this negation, an adaptation from [29].

In imperative programming, it is often the case that the **else**-part is omitted, which corresponds to "**else** do nothing." To capture this, we simply remove action c from (4), Thus, the statement **if** $\Diamond a$ **then** b **fi** can be expressed as follows:

$$\begin{array}{rcl} \mathit{if\_a\_b} & \longleftarrow & (\Diamond a) \otimes b \\ \mathit{if\_a\_b} & \longleftarrow & \Box \neg a \end{array} \tag{5}$$

Similarly, the following rules express a **while-do** statement:

$$\begin{array}{rcl} \mathit{while\_a\_b} & \longleftarrow & (\Diamond a) \otimes b \otimes \mathit{while\_a\_b} \\ \mathit{while\_a\_b} & \longleftarrow & \Box \neg a \end{array} \tag{6}$$

Intuitively, $?-\ \mathit{while\_a\_b}$ says, "while it is possible to do a, do b." Here, $\mathit{while\_a\_b}$ is a new proposition whose definition is *recursive,* which is what achieves the iterative effect. Notice, again, the role of "$\Box\neg a$" in the second clause of (6). Here it says that if a cannot be executed, then do nothing, which effectively terminates the loop. As with the **if-then-else** construct, it is suggestive to write **while** $\Diamond a$ **do** b **od** for proposition $\mathit{while\_a\_b}$, provided that $\mathit{while\_a\_b}$ does not occur in the head of any other rule.

Note that if b cannot be executed during an iteration, then the entire loop fails, so all previous iterations are undone. This is a form of automatic error recovery. In many cases, however, it may be desirable to not undo previous iterations, but to proceed with the loop either by ignoring the failed execution of b or by invoking a designated *error-handling* routine. In $\mathcal{TR}$, this can be expressed thus:

while $\Diamond a$ **do**
    if b **then do nothing else** *error-handler* **fi**
od

Here, if b fails during any iteration, then the error-handling transaction is executed. The if statement itself denotes an atom, $\mathit{if\_b}$, defined by the following rules:

$$\begin{array}{rcl} \mathit{if\_b} & \longleftarrow & \Diamond a \\ \mathit{if\_b} & \longleftarrow & (\Box \neg a) \otimes \mathit{error\text{-}handler} \end{array} \tag{7}$$

3.7 Active Databases

This section shows how active database systems can be represented as transaction bases in $\mathcal{TR}$. This representation captures several sides of the problem: (i) specifying an application using so-called active rules, (ii) detecting events, and (iii) specifying the algorithmic internals of the active database system, i.e., the policy for executing triggered actions. Because of space limitations, we consider a fairly simple system, one in which actions are triggered by the invocation of transactions, which sets no priorities on the triggered actions, and which executes triggered actions immediately. It is not hard to program more sophisticated systems.

The use of database programming languages to model active database systems has been discussed in [21, 32]. In those attempts, the underlying semantics is denotational. There are two main reasons for using $\mathcal{TR}$ to specify active database features and implementations. First,

$\mathcal{TR}$ provides a complete formalization (including a model and a proof theory) for the behavior of the system. Second, $\mathcal{TR}$ has one underlying notation and semantics, which can describe behavior procedurally and in detail, or declaratively and at a high level.

We shall use the notation for active rules[4] suggested in Starburst [33]:[5]

$$\begin{array}{l} \textbf{define active rule } a\_rule \\ \textbf{when } event \\ \textbf{if } condition \\ \textbf{then } action \end{array} \tag{8}$$

We consider these active rules to be part of the transaction base. This particular active rule is given the name *a_rule*, and its intended meaning is that when the given *event* occurs, and provided that *condition* is true at that moment, the system should automatically execute *action*. This policy is known as *immediate coupling* of conditions and actions [26]. Later, we show how active rules like these can be programmed in $\mathcal{TR}$.

In this paper, we limit our attention to two kinds of events: commencement of a transaction and termination of a transaction. For each named transaction, $trans(X)$, these two events are represented by two atomic formulas: $trans\_start(X)$ and $trans\_done(X)$, respectively. Active rules that are triggered by such events will be executed just before *trans* starts or just after it terminates. Events specified in this way generalize those supported in actual systems [31, 34, 18], and are analogous to the idea of method invocation and method termination proposed in [8].

An Example

Consider an active database system that enforces the following constraint: after any salary increase or a salary cut, no staff member should earn more than 120% of his manager's salary. Whenever this constraint is violated, the company policy is as follows: repeatedly increase the manager's salary by 2% while simultaneously decreasing the staff member's salary by 1%, until the constraint is satisfied.

We shall first represent this salary control policy informally, using $\mathcal{TR}$ syntax mixed with Starburst-style notation for active rules. After this, we describe a general mechanism for implementing active rules in $\mathcal{TR}$.

Let us assume that the appropriate way to raise or cut a salary by a given percent is by executing transactions $raise(Empl, Percent)$ and $cut(Empl, Percent)$. (The exact definition of these transaction is immaterial for the present discussion.) These transactions can be invoked by the user explicitly, or they may be subtransactions of some other transaction that is explicitly invoked.

To enforce the salary policy for staff members, we add a pair of active rules to the transaction base. For clarity, we use deductive rules to define conditions and actions. The active rule and

[4] We use the qualifier *active* to emphasize the differences between active database rules and (unqualified) deductive database rules.

[5] Our motive for separating the condition from the action in an active rule is simply a concession to existing approaches. In $\mathcal{TR}$, however, there is no intrinsic need to make such a distinction.

the deductive rules are all added to the transaction base.

$$\begin{array}{l} \textbf{define active rule} \;\; raise\_policy \\ \textbf{when} \;\; raise\_done(E,P) \\ \textbf{if} \;\; salary\_condition(E,M) \\ \textbf{then} \;\; cut(E,1) \end{array} \tag{9}$$

$$\begin{array}{l} \textbf{define active rule} \;\; cut\_policy \\ \textbf{when} \;\; cut\_done(E,P) \\ \textbf{if} \;\; salary\_condition(E,M) \\ \textbf{then} \;\; raise(E,2) \end{array} \tag{10}$$

$$\begin{array}{rcl} salary\_condition(E,M) & \leftarrow & staff(E) \wedge manages(M,E) \wedge salary(E,S_{emp}) \\ & & \wedge\, salary(M,S_{mngr}) \wedge S_{emp} > 1.2 \times S_{mngr} \end{array}$$

Thus, after each invocation of the transaction $raise(E,P)$, rule (9) checks if the employee, E, is a staff member, and whether his new salary, S_{emp}, is too high relative to his manager's salary, S_{mngr}. If this condition is satisfied, the staff member's salary is lowered by 1%. This, in turn, may trigger a raise of a manager's salary by 2%.

Since rule (9) invokes the transaction *cut*, we have an implicit recursion: raising a salary triggers rule (9), which can cause salary cut, triggering rule (10). The latter may cause salary raise again, and so on. In this way, the manager's salary is repeatedly increased, and the staff member's salary is repeatedly cut, until the constraints on their salaries are satisfied.

Implementing Triggers in $\mathcal{TR}$

We now show how an active rule of the form (8) can be represented in $\mathcal{TR}$. Recall that the events associated with named transactions, *e.g.*, $trans(X)$, are represented by a pair of atoms, $trans\_start(X)$ and $trans\_done(X)$, which denote the start and end of transaction execution. The idea is to define these events as transactions that invoke the appropriate active rules just before and just after the execution of $trans(X)$.

Implementing these ideas in $\mathcal{TR}$ involves three steps. The first step is to associate the transaction, *trans*, with the events *trans_start* and *trans_done*. This is accomplished by replacing each occurrence of $trans(X)$ in the transaction base by the following formula:

$$trans\_start(X) \otimes trans(X) \otimes trans\_done(X) \tag{11}$$

This idea is common in active systems, as it makes events easy to detect.

The second step is to associate each event with the active rules that it triggers. In addition, we must specify the order (or orders) in which the active rules may fire. Let $event(X)$ denote an arbitrary event, i.e., either $trans\_start(X)$ or $trans\_done(X)$. Using rules like the following, $event(X)$ executes a sequence of active rules of the form (8):

$$\begin{array}{lcl} event(X) & \leftarrow & a\_rule_1(X) \otimes a\_rule_2(X) \otimes \cdots \otimes a\_rule_n(X) \\ \quad \ddots & & \\ event(X) & \leftarrow & a\_rule_n(X) \otimes a\_rule_{n-1}(X) \otimes \cdots \otimes a\_rule_1(X) \end{array} \tag{12}$$

Each of these rules specifies a set of active rules and an order in which they may execute. If the order of the active rules is immaterial, then the definition of $event(X)$ will have one rule for each permutation of the active rules. By selecting a particular subset of all permutations, one can specify the *conflict resolution* strategy employed by the system [34, 3]. (The reader should not be frightened by the prospect of having to write $n!$ rules. First, this can be done automatically. Second, $\mathcal{TR}$ has an operator, called *shuffle*, which makes this tiresome encoding much more succinct. Furthermore, the number of rules does not increase the cost of executing $event(X)$, since $\mathcal{TR}$'s proof theory selects just one rule non-deterministically.)

The third, and final, step in implementing the triggering mechanism is to represent each active rule as a $\mathcal{TR}$-rule. The Starburst-style rule (8) is then represented as follows:

$$a\_rule \leftarrow \textbf{if} \;\; condition \;\; \textbf{then} \;\; action \;\; \textbf{fi} \tag{13}$$

Note that the use of the **if-then** imperative construct from Section 3.6 is important for the correctness of our encoding of active rules in $\mathcal{TR}$. It ensures that if *condition* fails, then *a_rule* still succeeds but leaves the database unchanged. Although it is tempting to define *a_rule* using a simpler rule,

$$a\_rule \leftarrow condition \otimes action$$

this representation is incorrect. Indeed, *a_rule* thus defined would fail if *condition* fails. As a consequence, the *event*-transaction in (12) would fail too. This in turn would cause the entire transaction in (11) to fail. Obviously, this is not what is required of a trigger. In contrast, by defining *a_rule* as in (13), we achieve the effect that whenever a pre-condition of an active rule fails, the rule simply does not fire, but the *event*-transaction succeeds anyway.

The Example, Continued:

Returning to our example, each occurrence of $raise(E, P)$ in the transaction base is thus replaced by the following formula:

$$raise\_start(E, P) \otimes raise(E, P) \otimes raise\_done(E, P)$$

The start of a *raise* transaction triggers no actions; so the event $raise\_start(E, P)$ is defined by the following rule, which does nothing and always succeeds:

$$raise\_start(E, P) \longleftarrow$$

A similar transformation is done for the *cut* transaction. The completion of a *raise* or a *cut* transaction is more complex since it does trigger actions. We represent *raise_done* and *cut_done* as follows:

$$\begin{aligned} raise\_done(E, P) &\longleftarrow raise\_policy(E, P) \\ cut\_done(E, P) &\longleftarrow cut\_policy(E, P) \end{aligned}$$

To represent the active rules themselves, we use the following $\mathcal{TR}$ rules:

$$\begin{aligned} raise\_policy(E, P) &\longleftarrow \textbf{if } salary\_condition(E, M) \textbf{ then } cut(E, 1) \textbf{ fi} \\ cut\_policy(E, P) &\longleftarrow \textbf{if } salary\_condition(E, M) \textbf{ then } raise(M, 2) \textbf{ fi} \end{aligned}$$

3.8 Dynamic Constraints on Transaction Execution

Because transactions are defined on paths, it is possible to express a large variety of constraints on the way they execute. For instance, we can place conditions on the state of the database during transaction execution, or we may forbid certain sequences of states. We refer to such conditions as *path constraints*, or *dynamic constraints*. Such constraints are particularly well suited to areas such as planning and design, where it is common to place constraints on the way things are done. This section illustrates a variety of dynamic constraints expressible in $\mathcal{TR}$. These include temporal constraints in the style of James Allen [4], such as, "immediately after," "some time after," "during," "at the start of," and "at the end of."

There are several important problems related to constraints. One such problem, and the main subject of this section, is *constraint satisfaction*. That is, given a transaction and a constraint, we want to execute the transaction in such a way that it satisfies the constraint. For example, we might ask a robot to carry out a task while not entering restricted areas and not executing certain undesirable or dangerous sequences of action. In general, starting from the current database, we want to find *some* way of executing a transaction while satisfying constraints.

Constraint satisfaction problems are particularly easy to express in $\mathcal{TR}$ because they correspond to classical conjunction. That is, if ψ and ϕ are transaction formulas, then the formula $\psi \wedge \phi$ means, "Do transaction ψ in such a way that ϕ is satisfied along the entire execution path." The formula ϕ thus constrains the way in which ψ executes.

Constraints Based on Serial Conjunction:

Two types of path constraint naturally arise in $\mathcal{TR}$: those based on serial conjunction, and those based on serial implication. The former specify that something must be true *somewhere* on a path, and the latter specify that something must be true *everywhere* on a path. These two types of path constraint correspond roughly to two types of database integrity constraint: those based on existential quantification, and those based on universal quantification, respectively. This section gives examples of the former type of constraint. For instance, the following formula requests a robot to go to room A, passing through rooms A_1, A_2 and A_3 along the way:
$goto(roomA) \ \wedge \ go\_thru(roomA_1) \ \wedge \ go\_thru(roomA_2) \ \wedge \ go\_thru(roomA_3)$
where $go\_thru(X)$ is defined in terms of serial conjunction. The full paper elaborates on this idea.

Constraints Based on Serial Implication:

This section considers constraints based on the binary connectives "$\Leftarrow$" and "$\Rightarrow$", called *left serial implication* and *right serial implication*. These connectives are defined in terms of serial disjunction, $\oplus$, which is the dual of $\otimes$. In particular, the formula $\psi \Leftarrow \phi$ is defined to be $\psi \oplus \neg\phi$, and $\phi \Rightarrow \psi$ is defined to be $\neg\phi \oplus \psi$. Intuitively, the formula $\phi \Rightarrow \psi$ means that transaction ψ must come immediately after transaction ϕ; or more precisely, *whenever* ϕ occurs, then ψ occurs just *after* it. The formula $\psi \Leftarrow \phi$ is a kind of dual. It says that *whenever* ϕ occurs, then ψ must have occurred just *before* it.

Constraints based on serial implication constrain a transaction during every moment of its execution. For instance, we might want a robot to remain inside a particular region while executing a task. We can also put constraints on specific actions that the robot might take. For instance, we might request a robot to perform a series of actions subject to the following constraints: (*i*) Before leaving a room, turn off all the lights; (*ii*) After entering a room, turn on all the lights; (*iii*) Unlock the rifle before firing it; (*iv*) Lock and reload the rifle after firing it. In these examples, "before" and "after" mean "immediately before" and "immediately after," respectively. Serial implication expresses these two relations. For example, constraints (*iii*) and (*iv*) are expressed by the following two formulas, respectively:

$$unlock \Leftarrow shoot \qquad\qquad shoot \Rightarrow lock \otimes load$$

In addition to the temporal relations "immediately before" and "immediately after," $\mathcal{TR}$ can express many other temporal relations in the style of James Allen's theory of time intervals [4]. These relations include "some time before," "some time after," "during," "at the start of," "at the end of," etc. The full paper elaborates on these ideas.

4 Comparison with Other Works

As far as databases are concerned, we are not aware of any other declarative approach to updates that is as comprehensive as $\mathcal{TR}$. In particular, none of the works discussed below is capable of expressing constraints on the execution of complex transactions. Likewise, none of them can *seamlessly* accommodate hypothetical state transitions with transitions that actually commit; and, with the exception of [22], all of the works are limited to updating sets of ground atomic facts. A much more extensive comparison can be found in [10].

Winslett [35] did foundational work on the meaning of updates to general logical theories. Later, Grahne, Katsuno and Mendelzon [22, 19] axiomatized various theories of state transition and studied tractable cases of what we call "elementary state transitions." Our approach to state transitions is inspired by these results.

Manchanda and Warren [25] introduce Dynamic Prolog—a logic system where update transactions "work right," *i.e.*, when failed, they do not leave a residue in the database. Like $\mathcal{TR}$, their logic can be used to update views, and transactions can be nondeterministic. However, they distinguish between update predicates and query predicates—a drawback if we keep an eye on object-oriented applications, as explained in the introduction. Furthermore, bulk updates, constraints on transaction execution, and the insertion and deletion of rules cannot be expressed, due to the chosen semantics. In addition, the proof theory for Dynamic Prolog is impractical for carrying out updates, since one must know the final database state *before* inference begins. Apparently, realizing this drawback, Manchanda and Warren developed an interpreter for "executing" transactions. However, this interpreter is incomplete with respect to the model theory and, furthermore, it is not based on the proof theory of Dynamic Prolog. To a certain extent, it can be said that Manchanda and Warren have managed to formalize their intuition procedurally, but not as an inference system.

Naqvi and Krishnamurthy [27] extended Datalog with update operators, which were later incorporated in the LDL language. Since LDL is geared towards database applications, this extension has bulk updates, for which an operational semantics exists. Unfortunately, the model theory presented in [27] is somewhat limited. First, it matches the proposed execution model only in the propositional case, and so it does not cover bulk updates. Second, it is only defined for update-programs in which commutativity of elementary updates can be assumed. For sequences of updates in which this does not hold, the semantics turns out to be rather tricky and certainly does not qualify as "model theoretic." Third, the definition of "legal" programs in [27] is highly restrictive, making it difficult to build complex transactions out of simpler ones.

Abiteboul and Vianu developed a family of declarative update languages [2], including impressive results on complexity and expressibility. However, these languages lack several features that are present in $\mathcal{TR}$. First, they apply only to relational databases, not to arbitrary sets of first-order formulas. Thus, it is not possible to insert or delete rules from a deductive database. Second, there is no facility for constraining transaction execution. Indeed, transaction output is the only concern. Third, these languages are not part of a full-blown *logic*: arbitrary logical formulas cannot be constructed, and although there is an operational semantics, there is no model theory and no logical inference system. It is therefore unlikely that these languages have the flexibility to find applications in other domains, such as AI. Finally, these languages do not support transaction subroutines. This lack of subroutines is reflected in the data complexity of some of the languages: they are in PSPACE, whereas recursive subroutines require EXPTIME.

The works [28, 17] are related to [2] in that they all borrow much of their syntax from deductive databases and yet their semantics is operational (although inspired by logical model theory). As such, these languages are in a different league than $\mathcal{TR}$; they are also unsuitable for defining transaction subroutines, nested transactions, constraints, and for reasoning about actions.

Acknowledgments: Alberto Mendelzon provided us with many insights regarding updates of logic theories. Thanks to Ray Reiter for commenting on various aspects of $\mathcal{TR}$, especially on the issues related to the frame problem. Discussions with Gösta Grahne, Peter Revesz, Fangzhen Lin, Javier Pinto, Dimitris Lagouvardos, Jan Van den Bussche, and Roel Wieringa are also gratefully acknowledged.

References

[1] S. Abiteboul and A.J. Bonner. Objects and views. In *ACM SIGMOD Conference on Management of Data*, pages 238–247, Denver, Colorado, May 29–31 1991. ACM.

[2] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 240–250, New York, 1988. ACM.

[3] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 479–487. Morgan Kaufmann, San Francisco, CA, 1991.

[4] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, July 1984.

[5] F. Bancilhon. A logic-programming/Object-oriented cocktail. *SIGMOD Record*, 15(3):11–21, September 1986.

[6] T. Barsalou, N. Siambela, A.M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *ACM SIGMOD Conference on Management of Data*, pages 248–257, Denver, Colorado, May 29–31 1991. ACM.

[7] C. Beeri. New data models and languages—The challenge. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–15, San Diego, CA, June 1992. ACM.

[8] C. Beeri and T. Milo. A model for active object-oriented database. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 337–349. Morgan Kaufmann, San Francisco, CA, 1991.

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

[10] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-270, University of Toronto, April 1992. Revised: February 1994. *ftp://csri-technical-reports/270/report.ps.*

[11] A.J. Bonner and M. Kifer. Transaction logic: An (early) exposé. In V.S. Alagar, L.V.S. Lakshmanan, and F. Sadri, editors, *Proceedings of the Workshop on Formal Methods in Databases and Software Engineering*, Workshops in Computing, pages 1–23. Springer-Verlag, 1993. Keynote address. Workshop held 15–16 May, 1992, Montreal, Canada.

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

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

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

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

[16] W. Chen. Declarative specification and evaluation of database updates. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, volume 566 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, December 1991.

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

[18] N. Gehani and V. Jagadish. ODE as an active database: Constraints and triggers. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 327–336. Morgan Kaufmann, San Francisco, CA, 1991.

[19] G. Grahne and A.O. Mendelzon. Updates and subjunctive queries. Technical Report KRR-TR-91-4, CSRI, University of Toronto, July 1991.

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

[21] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 455–467. Morgan Kaufmann, San Francisco, CA, 1991.

[22] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 387–394, Boston, Mass., April 1991.

[23] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, May 1995.

[24] R. Krishnamurthy and S. Naqvi. Non-deterministic choice in Datalog. In *Proceedings of the 3-d Intl. Conference on Data and Knowledge Bases*, pages 416–424. Morgan-Kaufmann Publ., 1988.

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

[26] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *ACM SIGMOD Conference on Management of Data*, pages 215–224, New York, 1989. ACM.

[27] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 251–262, New York, March 1988. ACM.

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

[29] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[30] D. Sacca and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 205–217, New York, April 1990. ACM.

[31] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *ACM SIGMOD Conference on Management of Data*, pages 281–290, New York, 1990. ACM.

[32] J. Widom. A denotational semantics for the starbust production rule language. *ACM SIGMOD Record*, 21(3):4–9, 1992.

[33] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to starbust. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 275–285. Morgan Kaufmann, San Francisco, CA, 1991.

[34] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *ACM SIGMOD Conference on Management of Data*, pages 259–270, New York, 1990. ACM.

[35] M. Winslett. A model based approach to updating databases with incomplete information. *ACM Transactions on Database Systems*, 13(2):167–196, 1988.

A Appendix: Model Theory

This section makes the discussion in Section 2.3 precise.

A.1 Path Structures

In the definitions below, each path structure has a domain of objects and an interpretation for all function symbols, which are used to interpret formulas on every path in the structure.

Definition A.1 (Path Structures) Let $\mathcal{L}$ be a first-order language with function symbols in $\mathcal{F}$ and predicate symbols in $\mathcal{P}$. A *path structure*, $\mathbf{M}$, over $\mathcal{L}$ is a quadruple $\langle U, I_{\mathcal{F}}, N, I_{path}\rangle$ where

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

  We shall use $Struct(U, I_{\mathcal{F}})$ to denote the set of all usual first-order semantic structures over $\mathcal{L}$ of the form $\langle U, I_{\mathcal{F}}, I_{\mathcal{P}}\rangle$, where $I_{\mathcal{P}}$ is some mapping that interprets predicate symbols in $\mathcal{P}$ by relations on U.
- N is a non-empty set of *states,* where each state is a non-empty subset of $Struct(U, I_{\mathcal{F}})$. An element of N is called a state of the path structure, $\mathbf{M}$.

  A *path* of length k in $\mathbf{M}$ is any finite sequence of states, $\langle s_1, \ldots, s_k\rangle$ where $k \geq 1$ and $s_i \in N$.
- I_{path} is a mapping that assigns to every path in $\mathbf{M}$ a first-order semantic structure in $Struct(U, I_{\mathcal{F}})$, subject to the restriction that $I_{path}(\langle s\rangle) \in s$ for every state s. (Recall that s is a *set* of semantic structures.)

$\square$

The mapping I_{path} serves as a semantic link between transactions and paths: Given a path and a transaction formula, I_{path} determines whether the formula is true on the path (Definition A.2, below). The restriction that $I_{path}(\langle s\rangle) \in s$ guarantees that any path of length 1 (*i.e.*, a view of the database state) is a model of the underlying database. Note that for an arbitrary path, π, the semantic structure $I_{path}(\pi)$ is independent of the subpaths of π. Intuitively, this means that we know nothing about the relationship between transactions and their subtransactions. Such knowledge, when it exists, is encoded in the transaction base. It is therefore in the definition of satisfaction that paths and subpaths are related.

Before defining satisfaction, it is convenient to define path *splits*. Given a path, $\langle s_1, ..., s_n\rangle$, any state, s_i, on the path defines a split of the path into two parts, $\langle s_1, ..., s_i\rangle$ and $\langle s_i, ..., s_n\rangle$. If path π is split into parts γ and δ, then we write $\pi = \gamma \circ \delta$. Thus, γ is a prefix of π, and δ is a suffix of π.

As in classical logic, in order to define satisfaction for quantified formulas and open formulas, it is convenient to introduce variable assignments. A *variable assignment*, ν, is a mapping, $\mathcal{V} \longmapsto U$, that takes a variable as input, and returns a domain element as output. We extend the mapping from variables to terms in the usual way, *i.e.*, $\nu(f(t_1, \ldots, t_n)) = I_{\mathcal{F}}(f)(\nu(t_1), \ldots, \nu(t_n))$.

Definition A.2 (Satisfaction) Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, N, I_{path} \rangle$ be a path structure, let π be a path in $\mathbf{M}$, and let ν be a variable assignment. Then:

1. $\mathbf{M}, \pi \models_\nu p(t_1, \ldots, t_n)$ if and only if $I_{path}(\pi) \models^c_\nu p(t_1, \ldots, t_n)$, for any atomic formula $p(t_1, \ldots, t_n)$, where "$\models^c_\nu$" denotes classical satisfaction in first-order predicate calculus.
2. $\mathbf{M}, \pi \models_\nu \neg\phi$ if and only if it is not the case that $\mathbf{M}, \pi \models_\nu \phi$.
3. $\mathbf{M}, \pi \models_\nu \phi \vee \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ or $\mathbf{M}, \pi \models_\nu \psi$.
4. $\mathbf{M}, \pi \models_\nu \phi \wedge \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ and $\mathbf{M}, \pi \models_\nu \psi$.
5. $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ if and only if $\mathbf{M}, \gamma \models_\nu \phi$ and $\mathbf{M}, \delta \models_\nu \psi$ for *some* split $\gamma \circ \delta$ of path π.
6. $\mathbf{M}, \pi \models_\nu \phi \oplus \psi$ if and only if $\mathbf{M}, \gamma \models_\nu \phi$ or $\mathbf{M}, \delta \models_\nu \psi$ for *every* split $\gamma \circ \delta$ of path π.
7. $\mathbf{M}, \pi \models_\nu (\exists X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *some* variable assignment μ that agrees with ν everywhere except on X.
8. $\mathbf{M}, \pi \models_\nu (\forall X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *every* variable assignment μ that agrees with ν everywhere except on X.

To do hypothetical reasoning, we add the following two items, where $\langle s \rangle$ is a path of length 1 containing state s:

9. $\mathbf{M}, \langle s \rangle \models_\nu \Diamond\phi$ if and only if there is a path, π, starting at state s, such that $\mathbf{M}, \pi \models_\nu \phi$ holds.
10. $\mathbf{M}, \langle s \rangle \models_\nu \Box\phi$ if and only if for every path, π, starting at state s, it is the case that $\mathbf{M}, \pi \models_\nu \phi$.

As in classical logic, the mention of variable assignment can be omitted for *sentences*, *i.e.*, for formulas with no free variables. From now on, we will deal only with sentences, unless explicitly stated otherwise. $\Box$

In $\mathcal{TR}$, atoms like $p(t_1, \ldots, t_n)$ play the role of "subroutine calling sequences" in programming-language parlance. Intuitively, executing the subroutine corresponds to finding a path on which $p(t_1, \ldots, t_n)$ is true. Items 5 and 6 establish a relationship between a path and its subpaths, which corresponds to the relationship between a transaction and its subtransactions. In particular, an atom, p, may be true on a path but false on all proper subpaths (and vice-versa). Intuitively, this means that transaction p does not call itself recursively.

Definition A.3 (Models of Transaction Formulas) A path structure, $\mathbf{M}$, is a *model* of a $\mathcal{TR}$-formula ϕ, denoted $\mathbf{M} \models \phi$, if and only if $\mathbf{M}, \pi \models \phi$ for every path π in $\mathbf{M}$. A path structure is a model of a set of formulas if and only if it is a model of every formula in the set. $\Box$

As usual in first-order logic, we define $\psi \leftarrow \phi$ and $\phi \rightarrow \psi$ to mean $\psi \vee \neg\phi$, and $\psi \leftrightarrow \phi$ to mean $(\phi \leftarrow \psi) \wedge (\phi \rightarrow \psi)$. By replacing $\vee$ with $\oplus$ (the dual of $\otimes$), we obtain another interesting pair of serial connectives: *left serial implication*, $\psi \Leftarrow \phi$, standing for $\psi \oplus \neg\phi$, and *right* serial implication, $\phi \Rightarrow \psi$, standing for $\neg\phi \oplus \psi$. Intuitively, these formulas say that, "action ϕ must be immediately preceded (resp., followed) by action ψ." Unlike "$\leftarrow$" and "$\rightarrow$", these connectives are not identical, *i.e.*, $\psi \Leftarrow \phi$ is *not* equivalent to $\phi \Rightarrow \psi$; rather, $\psi \Leftarrow \phi$ is equivalent to $\neg\psi \Rightarrow \neg\phi$. It is easy to verify that the following formulas, analogous to De Morgan's laws, are tautologies:

$$
\begin{array}{rcl}
\neg(\phi \oplus \psi) & \leftrightarrow & \neg\phi \otimes \neg\psi \\
\neg(\phi \otimes \psi) & \leftrightarrow & \neg\phi \oplus \neg\psi \\
(\phi \vee \psi) \otimes \eta & \leftrightarrow & (\phi \otimes \eta) \vee (\psi \otimes \eta) \\
(\phi \wedge \psi) \oplus \eta & \leftrightarrow & (\phi \oplus \eta) \wedge (\psi \oplus \eta) \\
(\phi \wedge \psi) \otimes \eta & \longrightarrow & (\phi \otimes \eta) \wedge (\psi \otimes \eta) \\
(\phi \vee \psi) \oplus \eta & \longleftarrow & (\phi \oplus \eta) \vee (\psi \oplus \eta)
\end{array}
\tag{14}
$$

Definition A.3 tells us what it means for a path structure to be a model of a transaction formula ϕ. Such formulas are used to define complex transactions in terms of simpler ones. In addition, we must define what it means to be a model of an elementary state transition, $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$. Informally, this transition states that u is an update that changes database $\mathbf{D}_1$ into database $\mathbf{D}_2$. The next two definitions make this idea precise.

Definition A.4 (Correspondence) Let $\mathbf{M} = \langle U, \mathcal{F}, N, I_{path} \rangle$ be a path structure. For each first-order formula $\mathbf{D}$, the expression $\mathbf{D} \rightsquigarrow s$ means, "s is the set of *all* (first-order) models of $\mathbf{D}$ in $\mathit{Struct}(U, I_{\mathcal{F}})$." We say that s *corresponds* to database $\mathbf{D}$. □

Note that the meaning of $\mathbf{D} \rightsquigarrow s$ depends on the path structure $\mathbf{M}$ (*i.e.*, on its domain and its interpretation of function symbols). This structure will always be clear from the context.

Definition A.5 (Models of Transition Bases) Let $\mathbf{M}$ be a path structure, let $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ be an elementary state transition, and suppose that $\mathbf{D}_1 \rightsquigarrow s_1$ and $\mathbf{D}_2 \rightsquigarrow s_2$. Then, the transition is *satisfied* in $\mathbf{M}$, denoted $\mathbf{M} \models \langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$, if and only if s_1 and s_2 are states of $\mathbf{M}$ and $\mathbf{M}, \langle s_1, s_2 \rangle \models u$. □

In this definition, $\langle s_1, s_2 \rangle$ is a path of length 2, and the entailment relation "$\models$" is from Definition A.2. Informally, the statement $\mathbf{M} \models \langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ means that u is true on the arc from s_1 to s_2 in $\mathbf{M}$. Note that unlike transaction formulas, which are true on paths, truth of an elementary transition is determined with respect to the entire path structure. We say that $\mathbf{M}$ is a *model* of a transition base $\mathcal{B}$ if and only if $\mathbf{M}$ satisfies every elementary state transition in $\mathcal{B}$.

A.2 Execution as Entailment

We are now ready to define *executional entailment*, a concept that connects the model theory with transaction execution. Informally, execution of formulas corresponds to truth on a path.

A $\mathcal{TR}$ program consists of three distinct parts: a transaction base $\mathbf{P}$, a database $\mathbf{D}$, and a transition base $\mathcal{B}$. Each of these parts plays a distinct role in defining executional entailment. Of these three parts, only the database is updatable. The other two parts specify transactions that update the database and/or answer queries. The transition base defines elementary updates (state transitions), and the transaction base contains logical rules that define complex queries and transactions. The transaction base will normally be composed of formulas containing the serial connectives $\otimes$ or $\oplus$, though classical first-order formulas are also allowed. In contrast, the database consists entirely of classical first-order formulas.

Definition A.6 (Executional Entailment) Let $\mathcal{B}$ be a transition base, and $\mathbf{P}$ be a transaction base. Let ϕ be a transaction formula, and let $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ be a sequence of databases (first-order formulas). Then, the following statement

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi \tag{15}$$

is true if and only if for every model, $\mathbf{M}$, of $\mathcal{B}$ and $\mathbf{P}$, there is a path $\langle s_0, s_1, \ldots, s_n \rangle$ in $\mathbf{M}$ such that $\mathbf{D}_i \leadsto s_i$, for $i = 0, 1, ..., n$, and $\mathbf{M}, \langle s_0, s_1, \ldots, s_n \rangle \models \phi$. Related to this are the following statements:

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \models \phi \tag{16}$$

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \mathbf{D}_n \models \phi \tag{17}$$

$$\mathcal{B}, \mathbf{P}, \text{---} \mathbf{D}_n \models \phi \tag{18}$$

which are true if and only if there is a sequence of databases $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ such that Statement (15) is true. $\Box$

Informally, Statement (15) says that a successful execution of transaction ϕ can change the database from state $\mathbf{D}_0$ to $\mathbf{D}_1 \ldots$ to $\mathbf{D}_n$. Formally, it means that every model of $\mathcal{B}$ and $\mathbf{P}$ has a path corresponding to $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ that satisfies formula ϕ. The statement is read as follows: "Under transition base $\mathcal{B}$ and transaction base $\mathbf{P}$, transaction ϕ may transform database $\mathbf{D}_0$ into database $\mathbf{D}_n$ by passing through intermediate states $\mathbf{D}_1, \ldots, \mathbf{D}_{n-1}$."

Normally, users issuing transactions know only the initial database state, $\mathbf{D}_0$; so defining transaction execution via (15) is not quite appropriate. To account for this situation, the version of entailment in (16) allows us to omit the intermediate and the final database states. Intuitively, Statement (16) says that transaction ϕ can execute successfully starting from database $\mathbf{D}_0$. Formally, this statement is read as follows: "Under transition base $\mathcal{B}$ and transaction base $\mathbf{P}$, transaction ϕ succeeds from database $\mathbf{D}$." When the context is clear, we simply say that transaction ϕ *succeeds*. Likewise, when statement (16) is not true, we say that transaction ϕ *fails*. In [12, 10], we present an inference system that allows us to actually *find* a database sequence $\mathbf{D}_1, \ldots, \mathbf{D}_n$ (in fact, to enumerate all sequences) that satisfy Statement (15) whenever a transaction succeeds.

Statement (18) is the dual of (16). Informally, it says that ϕ can execute successfully *terminating* at state $\mathbf{D}_n$. This is discussed more fully in [10]. Statement (17) is also a useful abbreviation. Intuitively, it says that ϕ can execute successfully starting at state $\mathbf{D}_0$ and ending at state $\mathbf{D}_n$.

The following lemma lists some straightforward consequences of Definition A.6.

Lemma A.1 (Basic Properties of Executional Entailment) *For any transition base $\mathcal{B}$, any transaction base* $\mathbf{P}$*, any database sequence* $\mathbf{D}_0, \ldots, \mathbf{D}_n$*, and any transaction formulas α and β, the following statements are all true:*

1. *If* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$ *and* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \wedge \beta$.

2. *If* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_i \models \alpha$ *and* $\mathcal{B}, \mathbf{P}, \mathbf{D}_i, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \otimes \beta$.

3. *If* $\alpha \leftarrow \beta$ *is in* $\mathbf{P}$ *and* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$.

4. *If* $\langle \mathbf{D}_0, \mathbf{D}_1 \rangle \alpha$ *is in $\mathcal{B}$, then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models \alpha$.

5. *If* $\mathbf{D}_0 \models^c \psi$ *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \psi$, *where ψ is a first-order formula, and $\models^c$ denotes classical entailment.*

Note that assertions in Lemma A.1 deal with inference of two kinds of true statements. On the one hand, items 4 and 5 infer truth directly from the transition base and the database. Specifically, item 4 deals with elementary updates, and item 5 handles database queries. On the other hand, items 1, 2 and 3 combine existing entailments to infer new truths. Specifically, item 1 infers classical conjunctions; item 2 infers serial conjunctions; and item 3 infers defined transactions. Items 2–5 anticipate the proof procedures given in [12, 10], and indeed, they form the model-theoretic basis of the procedures. Item 1 is the basis for a wide class of dynamic constraints, such as those in Section 3.8.

In Lemma A.1, $n = 0$ corresponds to the special case in which a transaction does not affect a database, *i.e.*, in which it acts as a query. In this case, classical and serial conjunction are identical. This is reflected by the following two rules, which are special cases of items 1 and 2, above, respectively:

1b. *If* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \alpha$ *and* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \beta$ *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \alpha \wedge \beta$.

2b. *If* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \alpha$ *and* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \beta$ *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \alpha \otimes \beta$.

In fact, we have the following lemma, which follows directly from the definitions. Informally, this lemma says that the result of evaluating a conjunctive query is the same whether the conjuncts are evaluated sequentially or in parallel.

Lemma A.2 (Conjunctive Queries) *For any transition base $\mathcal{B}$, any transaction base* $\mathbf{P}$*, any database* $\mathbf{D}$*, and any transaction formulas α and β,*

$$\mathcal{B}, \mathbf{P}, \mathbf{D} \models \alpha \wedge \beta \quad \textit{if and only if} \quad \mathcal{B}, \mathbf{P}, \mathbf{D} \models \alpha \otimes \beta$$