Transaction Logic: An (Early) Exposé*

Anthony J. Bonner† and Michael Kifer‡

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

Abstract

This paper is an informal account of *Transaction Logic*—a new logic recently introduced in [4]. Transaction Logic was devised to deal with the phenomena of state changes in logic programming, databases, and AI. Transaction Logic has a natural model theory and a sound and complete proof theory. Unlike many other logics, however, it is suitable for *programming* transactions, transactions that accomplish state transitions in a logically correct manner. Transaction logic amalgamates such features as hypothetical *and* committed updates, dynamic constraints on transaction execution, nondeterminism, and bulk updates. Transaction Logic also appears to be suitable 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.

*Keynote address in *Proceedings of the Workshop on Formal Methods in Databases and Software Engineering*, Springer-Verlag, Workshops in Computing Series, May 1993. Held, 15–16 May 1992, Montreal, Canada.

†email: bonner@db.toronto.edu. This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada and a Connaught Grant from the University of Toronto.

‡email: kifer@cs.sunysb.edu. This work was supported in part by NSF grant CCR-9102159 and a grant from New York Science and Technology Foundation. Work done while on sabbatical leave from Stony Brook University. This author wishes to thank Computer Systems Research Institute of University of Toronto for their support. Present address: Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, U.S.A.

1 Introduction

We offer an informal introduction to a novel logic, called Transaction Logic (abbreviated $\mathcal{TR}$), that is specifically designed to deal with the updating of arbitrary logical theories, especially 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 an arbitrary logical theory. 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. The Horn version is of particular interest since it allows transactions to be defined as logic-programs. This paper presents the model theory and proof theory of $\mathcal{TR}$, and develops many of its applications.

$\mathcal{TR}$ was designed with several application 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. The full paper, [4], has an extensive discussion of a number of applications of $\mathcal{TR}$, both practical and theoretical. We outline several of these applications here.

1. $\mathcal{TR}$ provides a logical account for many update-related phenomena. For instance, in logic programming, $\mathcal{TR}$ provides a logical treatment of the assert and retract operators in Prolog. This 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 [10], to provide a logical account of *methods*, that is, of programs that reside "inside" objects and update their 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 and design. STRIPS-like actions[1] and many aspects of hierarchical and non-linear planning are easily expressed in $\mathcal{TR}$. Although there have been previous attempts to give these phenomena declarative semantics, until now there has been no unifying *logical* framework that accounts for them all. $\mathcal{TR}$ provides just such a framework.

2. $\mathcal{TR}$ is a full-fledged logic. As such, it is more flexible and expressive than procedural systems in specifying transactions. Like procedural languages, $\mathcal{TR}$ is a language where simple actions can be combined into complex ones; but in $\mathcal{TR}$, actions can be combined in a greater variety of ways. In procedural languages, sequential composition is the only combinator, whereas in $\mathcal{TR}$, each logical operator combines actions in its own way. The result is that $\mathcal{TR}$ can specify transactions at many levels of detail, from the procedural to the declarative. At one extreme, the user may spell out an exact sequence of operations in excruciating detail. At the other extreme, he may specify loose constraints that the transaction must satisfy. In general, sequences and constraints can be arbitrarily mixed, and in this way, procedural and declarative knowledge are seamlessly integrated.

[1] STRIPS was an early AI planning system that simulated the actions of a robot arm [6].

3. Because of its generality, $\mathcal{TR}$ supports a wide range of functionality in several areas. This functionality includes database queries and views; unification and rule-base inference; transaction and subroutine definition; deterministic and non-deterministic actions; static and dynamic constraints; hypothetical and retrospective transactions; and a wide class of tests and conditions on actions, including pre-conditions, post-conditions, and intermediate conditions. Furthermore, many problems related to the updating of incomplete information can be dealt with efficiently. Perhaps most important, these diverse capabilities are *not* built into $\mathcal{TR}$ separately. Rather, they all derive from a small number of basic ideas embodied in a single logic.

4. Commercial database systems are poor at specifying transactions. For example, in standard SQL, one can only define relatively simple updates, and even then, one must often abandon the relational algebra and resort to relatively awkward subqueries. More seriously, one cannot define transaction procedures in SQL. The main reason for this is that there is no way to combine simple transactions into more complex ones. This limitation should be contrasted with SQL's elegant view mechanism, in which views can easily be combined to produce other views. The result is that to define transactions, a user must go outside of SQL and embed it within a procedural language, such as C. A problem with this approach is that embedded SQL is an order of magnitude more difficult to use than pure SQL. In addition, embedded SQL is much less amenable to type checking and query optimization than SQL. Actually, it should not be surprising that difficult problems arise in trying to define updating transactions, since relational databases are founded on relational algebra (or, equivalently, first-order logic), which is a language for expressing *queries*, not updates. A solution, then, is to develop a logic that can serve as a foundation for queries *and* transactions. $\mathcal{TR}$ is just such a logic.

5. For a wide class of problems, $\mathcal{TR}$ avoids what is known in AI as the *frame problem*. The frame problem arises because, to reason about updates, one must specify what does *not* change, as well as what does. Many ingenious solutions have been proposed to deal with the frame problem, but most of them have aimed at the very general case of *arbitrary* reasoning about updates to arbitrary logical theories. As such, the proposed solutions have been conceptually complex and computationally intractable, even for simple updates. Fortunately, many interesting and important problems do not require a completely general solution to the frame problem. $\mathcal{TR}$, for example, was designed so that the frame problem does not arise when actions are *executed* or *planned*. This is possible for two reasons. First, transactions in $\mathcal{TR}$ are specified in terms of a small set of basic updates. Once the frame problem is solved for these updates, the solution propagates automatically to complex actions. Thus, $\mathcal{TR}$ programmers do not experience the frame problem, just as conventional programmers do not. In addition, the proof theory for $\mathcal{TR}$ actually updates the database during inference, just as procedural systems update the database during execution. This contrasts sharply with other logical formalisms, like the situation calculus, which reason about updates, but never actually change the database. In this way, $\mathcal{TR}$ provides simple, efficient handling of the frame problem, for both the specification and execution of

transactions.

On the surface, there would appear 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, database researchers continue to complain that there is no clear declarative semantics for database updates [2, 1]. In fact, no action logic has ever been adopted by a database or logic programming system, and none has become a core of database or logic-programming theory. This is in stark contrast to classical logic, which is the foundation of database queries and logic programming, both in theory and in practice.

There appear to be a few simple reasons for this unsuitability of existing action logics.

One major problem is that most logics of time or action are *hypothetical*, that is, they do not permanently change the database. Instead, these logics reason about what *would* happen *if* certain sequences of actions took place. For instance, they might infer that *if* the pawn took the knight, then the rook *would* be threatened. Such systems are useful for reasoning about alternatives, but they do not actually *accomplish* state transitions. Hence, these logics do not provide executable specifications. Contrast this with $\mathcal{TR}$, a logic where *both* real and hypothetical updates are possible. Procedures in $\mathcal{TR}$ may commit their updates (and thus permanently change the database), reason about these updates without committing them, or they may do any combination thereof.

Another major problem is that most logics of time or action were not designed for programming. Instead, they were intended for specifying *properties* of programs and for reasoning about them. In such systems, one might say that action A precedes action B, and that B precedes C, from which the system would infer that A precedes C. Thus, these formalisms are useful for *reasoning about* actions, but are not that useful for *specifying* what the actions A, B and C actually do, and even less so for executing them.

A related problem is that many logics of action cannot assign names to composite transactions. In their intended context (of reasoning about sequences of actions), this is not a shortcoming. However, this renders such logics inappropriate for programming transactions, since specifying transactions without a naming facility is like programming without subroutines. From a programmer's standpoint, the lack of a straightforward naming facility defeats the purpose of using logic in the first place, which is to free the user from the drudgery of low-level details.

Another problem is that many logics (and all relational databases) make a clear distinction between non-updating queries and actions with side effects. However, this distinction is blurred in object-oriented systems, where both queries and updates are special cases of a single idea: method invocation (or message passing). In such systems, an update can be thought of as a query with side effects. In fact, every method is simply a *program* that operates on the data. $\mathcal{TR}$ models this uniformity naturally, treating all methods equally, thereby providing a logical foundation for object-oriented databases.

The important point here is that, unlike most formalisms of action, $\mathcal{TR}$ is not *forced* to distinguish between updates and queries; but the distinction always exists as an *option*. This is possible because an application can introduce logical sorts, one for queries, and one for updates. This is comparable to the distinction in deductive databases between base predicates and derived predicates. Even when an application makes such distinctions, $\mathcal{TR}$ treats all predicates uniformly, just as classical logic does. Most other logics are not capable of this uniform treatment, since they see queries as propositions, but updates (or actions) are seen as entities of a different, incompatible nature. For instance, in situation calculus [12], actions are function terms, while in Dynamic and Process Logics [7, 8], actions are modal operators.

Although $\mathcal{TR}$ is different from logics of action, it is comparable to declarative query languages, like Prolog and SQL. In Prolog, for instance, one does not reason about logic programs. Instead, one specifies and executes them. Thus, given a logic program and a database of atomic facts, one can ask, "Is $p(a)$ true?", but one cannot ask not, "Will $p(a)$ *always* be true, i.e., for *every* database?" Logics of action are aimed at the latter kind of question (applied to actions); $\mathcal{TR}$ is aimed at the former, but without sacrificing the ability to handle the latter.

The system that comes closest in spirit to $\mathcal{TR}$ is Prolog. Although Prolog is not a logic of action *per se*, transactions can be defined in Prolog via the operators *assert* and *retract*. Prolog transactions have some of the properties that we want to model: (*i*) updates are real, not hypothetical; (*ii*) named procedures can be composed from simpler procedures; (*iii*) all predicates can have both a truth value and a side effect on the database; and (*iv*) the frame problem does not arise. Unfortunately, updates in Prolog are non-logical operations. Thus, each time a programmer uses "assert" or "retract" operators, he moves further away from declarative programming. In addition, Prolog has the opposite problem *vis a vis* most action logics: updates are always committed and cannot be rolled back. It is therefore not possible for a Prolog update to have post-conditions. For instance, one cannot perform an update tentatively, test its outcome, and then commit the update only if some condition is met. Due to these drawbacks, transactions are often the most awkward of Prolog programs, and the most difficult to understand, debug, and maintain.

In addition, updates in Prolog are not integrated into a complete logical system. It is therefore not clear how *assert* and *retract* should interact with other logical operators such as disjunction and negation. For instance, what does $assert(X) \vee assert(Y)$ mean? or $\neg assert(X)$? or $assert(X) \leftarrow retract(Y)$? Also, how does one logically account for the fact that the order of updates is important? Finally, what does it mean to update a database that contains arbitrary logical formulas? None of these questions is addressed by Prolog's classical semantics or its update operators.

$\mathcal{TR}$ provides a general solution to the aforementioned limitations, both of Prolog and of action logics, by providing a syntax and a semantics in which updates can be combined with logical operators to build a large number of interesting and useful formulas.

Syntactically, $\mathcal{TR}$ extends first-order logic with a single binary operator, $\otimes$, which we call *serial conjunction*. Intuitively, the formula $\psi \otimes \phi$ means, "First execute transaction ψ, and then execute transaction ϕ." Serial conjunction may be combined arbitrarily with the standard logical operators, such as $\wedge$, $\vee$ and $\neg$, to build a wide variety of formulas, such as $\neg[b(X) \otimes c(X)]$ and $\forall X[a(X) \leftarrow b(X) \otimes c(X)]$.

Semantically, $\mathcal{TR}$ is related to Process Logic [8], but is different from it in several important respects. As in Process Logic, a model in $\mathcal{TR}$ consists of a set of states, and actions cause transitions from one state to another. In fact, an action may cause a sequence of transitions, passing from an initial state, through intermediate states, to a final state. Like Process Logic, formulas in $\mathcal{TR}$ are *not* evaluated at states. Instead, they are evaluated on *paths*, which are sequences of states. This property enables $\mathcal{TR}$ to express a wide range of constraints on transaction execution. Unlike Process Logic, however, $\mathcal{TR}$ does not distinguish between programs and propositions: In $\mathcal{TR}$, every formula is both a proposition and a program. This is a natural way to model database transactions, since a transaction can both change the database and—as a query—return an answer. Thus, pure queries and pure updates are merely two extremes on a spectrum of possible transactions. In fact, in $\mathcal{TR}$, a single notion of truth subsumes two ideas: (i) the classical concept of a truth value, and (ii) the hitherto non-logical concept of a "side effect" on the database. This uniformity renders $\mathcal{TR}$ suitable for a number of diverse applications in areas such as relational databases, object-oriented databases, logic programming, and Artificial Intelligence. More discussion of the relationship between $\mathcal{TR}$ and process logics appears in [4].

Like classical logic, $\mathcal{TR}$ has a "Horn" version that is of particular interest for logic programming. In Horn $\mathcal{TR}$, a transaction is defined by Prolog-style rules in which the premise specifies a *sequence* of queries and updates. Furthermore, just as $\mathcal{TR}$ is an extension of classical first-order logic, Horn $\mathcal{TR}$ is an extension of classical Horn-clause logic.

Horn $\mathcal{TR}$ has a clean and simple proof theory that is sound and complete with respect to its model theory. This proof theory is described in detail in [4]. Two versions are presented, a bottom-up version based on natural deduction, and a top-down version based on resolution. As in classical Horn logic, the resolution-based proof theory is probably the one that would be used in practice. The proof theory for Horn $\mathcal{TR}$ is much simpler than the proof theory for full $\mathcal{TR}$, which involves encoding $\mathcal{TR}$ in a more general logic of state transitions. The latter will be discussed in a forthcoming report [3].

2 Overview and Introductory Examples

From the user's point of view, using $\mathcal{TR}$ is similar to using Prolog or using a relational database system. That is, the user may specify rules and he may pose queries and request updates. In $\mathcal{TR}$, the user sees no obvious or immediate difference between queries and updates. An update is just a query with side effects (which can be detected only by issuing subsequent queries). In general, the user issues transactions, and the system responds by displaying answers and updating the database. This section provides simple examples of how this behavior appears to the user and how it is described formally. The examples also illustrate several dimensions of $\mathcal{TR}$'s capabilities.

One of these capabilities, *non-deterministic transactions,* should be mentioned at the outset. Non-determinism has applications in many areas, but it is especially well-suited for advanced applications, such as those found in Artificial Intelligence. For instance, the user of a robot simulator might instruct the robot to build a stack of three blocks, but he may not say (or care) which three blocks to use. Likewise, the user of a CAD system might request the system to run an electrical line from one point to another, but without fixing the exact route, except in the form of loose constraints (*e.g.*, do not run the line too close to wet or exposed areas). These are the kinds of non-deterministic transactions that $\mathcal{TR}$ can specify. In such transactions, the final state of the database is non-deterministic, as it depends on choices made by the system at run time. $\mathcal{TR}$ enables users to specify what choices are allowed. When a user issues a non-deterministic transaction, the system makes particular choices (which may be implementation-dependent), putting the database into one of the allowed new states.

For all but the most elementary applications, a transaction execution will be characterized not just by an initial and a final state, but rather by a sequence of *intermediate* states that it passes through. For example, as a robot simulator piles block upon block upon block, the transaction execution will pass from state to state to state. Like the final state, intermediate states may not be uniquely determined at the start of the execution. For example, the robot may have some (non-deterministic) choice as to which block to grasp next. We call such a sequence of database states the *execution path* of the transaction. $\mathcal{TR}$ represents execution paths explicitly. By doing so, it can express a wide range of constraints on transaction execution. For example, a user may require every intermediate state to satisfy some condition, or he may forbid certain sequences of states.

To describe the execution of transactions formally, we use statements of the following form, which express a form of logical entailment in $\mathcal{TR}$, called *executional entailment*:

$$\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \psi \tag{1}$$

Here, $\mathbf{P}$ and each $\mathbf{D}_i$ is a logical formula, as is ψ. Intuitively, $\mathbf{P}$ is a set of transaction definitions, ψ is a transaction invocation, and $\mathbf{D}_0, \ldots, \mathbf{D}_n$ is a sequence of databases, representing all the states of transaction execution. Statement (1) means that $\mathbf{D}_0, \ldots, \mathbf{D}_n$ is an execution path of transaction ψ. That is, if the current database state is $\mathbf{D}_0$, and if the user issues the transaction ψ (by typing $?-\ \psi$, as in Prolog), then the database *may* go from state $\mathbf{D}_0$ to state $\mathbf{D}_1$, to state $\mathbf{D}_2$, etc., until it finally reaches state $\mathbf{D}_n$, after which the transaction terminates. We emphasize the word "may" because ψ may be a non-deterministic transaction. As such, it may have many execution paths beginning at $\mathbf{D}_0$, possibly of different length. The proof theory for $\mathcal{TR}$ can derive each of these paths, but only one of them will be (non-deterministically) selected as the actual execution path; the final state, $\mathbf{D}_n$, of that path then becomes the new database.

Unlike other formalisms, $\mathcal{TR}$ does not draw a thick line between transactions and queries. In fact, any transaction, ϕ, that does not cause a state change can be viewed as a query. This state of affairs is formally expressed by the statement $\mathbf{P}, \mathbf{D}_0 \models \phi$, a special case of statement (1) in which $n = 0$. In this case, $\mathbf{D}_0$ is a sequence of databases of length 1. This uniform treatment of transactions and queries is crucial for successful adaptation of $\mathcal{TR}$ to the object-oriented domain,[2] because many object-oriented systems do not syntactically distinguish between state-changing methods and information-retrieving methods.

Several other aspects of statement (1) should be mentioned at this point. First, the transaction base $\mathbf{P}$ can be *any* formula in $\mathcal{TR}$. In practice, though, it will often be a

[2]by combining $\mathcal{TR}$ with F-logic [10], for example.

conjunction of Prolog-like rules, which we will represent as a finite set of formulas. In any case, regardless of its form, we call $\mathbf{P}$ a *transaction base*. As the name suggests, transaction bases define transactions and, as a special case, queries. The databases, $\mathbf{D}_1, \ldots, \mathbf{D}_n$, are also finite sets of formulas; however, they are classical first-order formulas. These formulas are *not* restricted to be ground atomic facts, and it is entirely possible for each $\mathbf{D}_i$ to contain Prolog-like rules. The difference between a database and a transaction base is that formulas in a transaction base may use full syntax of $\mathcal{TR}$, while formulas in a database are limited to the syntax of first-order logic, a subset of $\mathcal{TR}$.

Statements of the form (1) provide a formal view of all that is possible in the transaction system. However, this formal view is somewhat different from the user's view of the system. From his perspective, a $\mathcal{TR}$ logic program has two parts: a transaction base, $\mathbf{P}$, which the programmer provides,[3] and a *single* current database, $\mathbf{D}$, which he wishes to access and possibly modify. The transaction base is immutable; that is, it cannot be changed by other transactions. In contrast, the database constantly changes—it is updated when the user executes transactions defined in $\mathbf{P}$. In particular, if $\mathbf{D}$ contains Prolog-style rules, they, too, can be modified.

The rest of this section illustrates our notation and the capabilities of $\mathcal{TR}$ through a number of simple examples. The examples illustrate how $\mathcal{TR}$ uses logical operators to combine simple actions into complex ones. These operators include the standard ones, such as $\wedge$, $\vee$ and $\neg$, and a new operator of sequential composition, $\otimes$. For the purpose of illustration, our example databases are sets of ground atomic formulas, and insertion and deletion of atomic formulas are elementary update operators. However, $\mathcal{TR}$ is not restricted to this type of database nor to these particular elementary updates.

2.1 Simple Transactions

This subsection introduces some of the basic ideas of $\mathcal{TR}$. Starting with purely declarative queries, we show how $\mathcal{TR}$ extends classical logic to represent procedural phenomena, such as updates and action-sequences. We then show how these ideas can be combined to form more complex transactions. The examples of this subsection focus on simple, deterministic transactions, ones that a user might issue directly to a $\mathcal{TR}$ interpreter. Subsequent subsections illustrate more sophisticated transactions.

Since $\mathcal{TR}$ queries are a special kind of transaction, they are defined in the transaction base, $\mathbf{P}$, just as other transactions are. These definitions are similar to Horn rules in logic programming, and we refer to them as *serial-Horn* formulas. These formulas draw inferences from base facts just as logic programs and deductive databases do. In fact, if the transaction base consists entirely of first-order formulas, then entailment in $\mathcal{TR}$ reduces to first-order entailment. That is, if $\mathbf{P}$ and ϕ are both first-order formulas, then

$$\mathbf{P}, \mathbf{D} \models \phi \qquad \textit{iff} \qquad \mathbf{P} \wedge \mathbf{D} \models^c \phi$$

where $\models^c$ denotes classical entailment. In this way, classical logic—the medium in which

[3] The term "programmer" should be taken here in a broad sense. For instance, parts of the transaction base may be provided by the knowledge-base engineer, by the database administrator, or by any number of system or application programmers.

queries are traditionally expressed—is the starting point for $\mathcal{TR}$. Logic programs, deductive databases, and first-order knowledge-bases can thus be adopted for use in $\mathcal{TR}$ with minimal (if any) change. That is, they are upward compatible with $\mathcal{TR}$.

Example 2.1 (Classical Inference) Suppose that the database, $\mathbf{D}$, contains the atom q, and that the transaction base, $\mathbf{P}$, contains the rule $p \leftarrow q$. By typing $?-\ p$, the user is posing a query, asking the $\mathcal{TR}$ interpreter if p is true. In this case, p can be inferred from the contents of the database and the transition base, so the interpreter returns "true." Likewise, the interpreter returns "true" if the user types $?-\ q$, or if he types $?-\ p \wedge q$. Note that in each case, the database remains unchanged. We represent this behavior formally by the following three statements:

$$\mathbf{P}, \mathbf{D} \models p \qquad \mathbf{P}, \mathbf{D} \models q \qquad \mathbf{P}, \mathbf{D} \models p \wedge q$$

□

In $\mathcal{TR}$, all transactions are a combination of queries and updates. Queries do not change the database, and can be expressed in classical logic, as Example 2.1 shows. In contrast, updates do change the database, and are expressed in an extension of classical logic. We refer to the simplest kind of updates as *elementary*. In principle, there is no restriction on the changes that an elementary update can make to a database, though in practice, we expect them to be simple and cheap. In this overview, we use the insertion and deletion of atomic formulas as canonical examples of elementary updates. Other kinds of elementary updates, such as SQL-style *bulk* updates, can be found in [4].

Elementary updates are atomic in that they cannot be decomposed into simpler updates. We therefore represent them by atomic formulas. Like all atomic formulas, elementary updates have a truth value; but in addition, they have a side effect on the database. We represent this idea formally by statements of the following form:

$$\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \models u$$

This statement says that the atomic formula u is (the name of) an update that changes the database from state $\mathbf{D}_1$ to state $\mathbf{D}_2$. Although any atomic formula can be an update, it is a good programming practice to reserve a special set of predicate symbols for this purpose. For example, for each predicate symbol p, we shall use another predicate symbol *ins:p* to represent insertions into p. Likewise, we use the predicate symbol *del:p* to represent deletions from p.

Example 2.2 (Elementary Updates) If p is a binary predicate symbol, then the atoms $ins{:}p(a,b)$ and $del{:}p(a,b)$ are elementary updates. Intuitively, the atom $ins{:}p(a,b)$ means, "insert the atom $p(a,b)$ into the database," while $del{:}p(a,b)$ means, "delete $p(a,b)$ from the database." From the user's perspective, typing $?-\ ins{:}p(a,b)$ to the interpreter changes the database from $\mathbf{D}$ to $\mathbf{D} + \{p(a,b)\}$. Likewise, typing $?-\ del{:}p(a,b)$ changes the database from $\mathbf{D}$ to $\mathbf{D} - \{p(a,b)\}$. We express this behaviour formally by the following two statements, which are true for any transaction base $\mathbf{P}$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{p(a,b)\} \models \mathit{ins{:}p}(a,b)$$
$$\mathbf{P}, \mathbf{D}, \mathbf{D} - \{p(a,b)\} \models \mathit{del{:}p}(a,b)$$

□

Here we use the operator "+" to denote set union when it is applied to databases. This is possible because all examples in this paper will deal with databases that are sets (or conjunctions) of ground atomic formulas. Thus, the expression $\{p, q\} + \{r, s\}$ denotes the set $\{p, q, r, s\}$, which in turn stands for the formula $p \wedge q \wedge r \wedge s$. Likewise, we use the operator "$-$" for set difference. For instance, $\{p, q, r, s\} - \{q, s\}$ denotes $\{p, r\}$, which stands for $p \wedge r$. It is hoped that this notation will not mislead the reader into thinking that, say, $\mathbf{D} + \{p(a,b)\}$ above represents a model—it does not. In general, $\mathbf{D}$ can be a set of arbitrary first-order formulas.

We emphasize here that insertions and deletions are *not* built into the semantics of $\mathcal{TR}$. Thus, there is no *intrinsic* connection between the names p, *ins:p* and *del:p*. Our use of these names is merely a convention for purposes of illustration. In fact, p, *ins:p* and *del:p* are ordinary predicates of $\mathcal{TR}$, where the connection between them is established via axioms of the so called *transition base*, as explained later.

$\mathcal{TR}$ is not committed to any particular set of elementary updates. Indeed, we expect that each database system will want its own repertoire of elementary updates, tuned to the applications at hand. Within each database system, though, we expect this repertoire of updates to be relatively stable. Thus, for most users of a given system, it will *appear* as though there is a fixed set of elementary updates. The user's job is to combine elementary updates into complex transactions using the facilities provided by $\mathcal{TR}$.

A basic way of combining transactions is to sequence them, *i.e.*, to execute them one after another. For example, one might take money out of one account and then, if the withdrawal succeeds, deposit the money into another account. To combine transactions sequentially, we extend classical logic with a new binary connective, $\otimes$, which we call *serial conjunction*. The formula $\psi \otimes \phi$ denotes the composite transaction consisting of transaction ψ followed by transaction ϕ.[4] Unlike elementary updates, sequential transactions often have intermediate states, as well as initial and final states. We express this behavior formally by statements like the following:

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \models \psi \otimes \phi$$

This statement says that the transaction $\psi \otimes \phi$ changes the database from $\mathbf{D}_0$ to $\mathbf{D}_1$ to $\mathbf{D}_2$. Here, $\mathbf{D}_1$ is an intermediate state.

Example 2.3 (Serial Conjunction) The expression *ins:a* $\otimes$ *ins:b*, where a and b are ground atomic formulas, denotes a sequence of two insertions. This transaction means, "First insert a into the database, and then insert b." Thus, if the initial database is $\mathbf{D}$, and if the user

[4] Logics of action sometimes use a semicolon to denote serial conjunction. We have avoided this notation for two reasons: (i) in Prolog, a semicolon means disjunction, and (ii) there is no natural symbol to stand for the dual of the semicolon (as we shall see, this dual is a useful operator in its own right). In contrast, $\oplus$ naturally stands for the dual of $\otimes$. We can therefore rewrite $\neg(\psi \otimes \phi)$ as $\neg\psi \oplus \neg\phi$.

issues a transaction by typing $?-$ *ins:a* $\otimes$ *ins:b*, then during execution, the database will change from $\mathbf{D}$ to $\mathbf{D}+\{a\}$ to $\mathbf{D}+\{a,b\}$. We express this behaviour formally by the following statement, which is true for any transaction base $\mathbf{P}$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D}+\{a\}, \mathbf{D}+\{a,b\} \quad \models \quad \mathit{ins{:}a} \otimes \mathit{ins{:}b}$$

□

2.2 Tests and Conditions

Serial conjunction applies not only to updates, but to any transaction, including queries. We can therefore place queries that act as tests anywhere in a sequence of updates. This allows us to monitor the progress of a transaction, and force it to abort if certain conditions are not met.

In the simplest case, we can define pre-conditions by placing a query at the beginning of a transaction. For instance, before withdrawing money from an account, we should check that enough money is available. If there is, then the transaction can continue, and the database can be updated; if not, then the transaction should abort, and the database should remain in its original state.

Example 2.4 (Pre-Conditions) If the atom b is a query (*i.e.*, has no side effects), then the expression $b \otimes$ *ins:c* denotes a test followed by an update. Intuitively, it means, "check that b is true, and if so, insert c into the database." In other words, if b is true in the initial database state, then the transaction succeeds and the database is changed from $\mathbf{D}$ to $\mathbf{D}+\{c\}$; otherwise, the transaction fails and the database remains in the initial state, $\mathbf{D}$. Formally,

$$\begin{array}{ll} \textit{if} & \mathbf{P}, \mathbf{D} \models b \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D}+\{c\} \models b \otimes \mathit{ins{:}c} \end{array}$$

□

By placing tests at other points in a transaction, sophisticated behavior can be specified in a simple and natural way. For instance, tests at the end of a transaction act as post-conditions, which query the *final* state of the database. If these tests succeed, then the transaction commits and the database is permanently changed. Otherwise, if the tests fail, then the transaction aborts and the database is rolled back to its original state. Post-conditions are particularly useful for aborting transactions that have forbidden side effects. For instance, a move in chess is forbidden if it puts you into check. Likewise, changes to a circuit design may be forbidden if the new design violates certain conditions (*e.g.*, limits on cost, size, or power consumption). It is worth noting that post-conditions can be awkward, if not impossible to express in other formalism of action, such as the situation calculus [12, 15].

Example 2.5 (Post-Conditions) If the atom b is a query (*i.e.*, has no side effects), then the expression $ins{:}c \otimes b$ denotes an update followed by a test. Intuitively, it means, "insert c into the database, and then check that b is true." In other words, the database state is first changed from $\mathbf{D}$ to $\mathbf{D} + \{c\}$. If b is true in the final state, then the transaction succeeds and the change is committed; otherwise, the transaction fails and the database is rolled back to the initial state, $\mathbf{D}$. Formally,

$$\begin{array}{lll} \textit{if} & \mathbf{P}, \mathbf{D} + \{c\} & \models \; b \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{c\} & \models \; ins{:}c \otimes b \end{array}$$

□

2.3 Non-Deterministic Transactions

In addition to being sequenced, transactions in $\mathcal{TR}$ can be combined using any of the classical connectives. Two of these connectives, disjunction and existential quantification, can be used to build non-deterministic transactions. For instance, the formula $\psi \vee \phi$ means, "do transaction ψ or do transaction ϕ." Likewise, the formula $\exists X\ \phi(X)$ means, "do transaction $\phi(c)$ for some value c."

Example 2.6 (Classical Disjunction) If the user issues a transaction by typing $?-\ ins{:}a \vee ins{:}b$, then after execution, the final database will be either $\mathbf{D} + \{a\}$ or $\mathbf{D} + \{b\}$. In effect, the user has stated that he doesn't care which update is made, so the system chooses one of them and executes it. Formally, *both* of the following statements are true, for any transaction base $\mathbf{P}$:

$$\begin{array}{lll} \mathbf{P}, \mathbf{D}, \mathbf{D} + \{a\} & \models & ins{:}a \vee ins{:}b \\ \mathbf{P}, \mathbf{D}, \mathbf{D} + \{b\} & \models & ins{:}a \vee ins{:}b \end{array}$$

□

In addition to defining non-deterministic transactions, classical disjunction is also needed (in combination with negation) to define implication, exactly as in classical logic. Implication is illustrated later in this section.

When non-determinism results from classical disjunction, alternatives are fixed and independent of the database. However, when existential quantification is used, the number of alternatives can be infinite and data-dependent.

Example 2.7 (Existential Quantification) Suppose the user issues the transaction $?-\ ins{:}q(X)$ (or, equivalently, $?-\ \exists X\ ins{:}q(X)$). The database system will then insert a tuple, $\langle c \rangle$, into relation q non-deterministically. That is, after execution, the database state will be $\mathbf{D} + \{q(c)\}$, for some (unspecified) value c. Formally, the following statement is true for every value c in the Herbrand universe:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{q(c)\} \quad \models \quad \exists X\ ins{:}q(X)$$

□

Example 2.8 (Copying Tuples Non-Deterministically) Consider the transaction $?-\ \exists X\,[p(X) \otimes ins{:}q(X)]$, which requests the system to copy a tuple from p to q non-deterministically. The system will first non-deterministically choose a tuple, $\langle c \rangle$, from relation p, and then insert the tuple into relation q. Formally, for every value c,

$$\begin{array}{lll} \textit{if} & \mathbf{P}, \mathbf{D} & \models\ p(c) \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{q(c)\} & \models\ \exists X\,[p(X) \otimes ins{:}q(X)] \end{array}$$

For instance, suppose the database $\mathbf{D}$ contains the atoms $p(b)$, $p(c)$ and $p(d)$. Then there are three choices for X: b, c, and d. Each choice results in a different update, since either $q(b)$, $q(c)$, or $q(d)$ will be inserted into $\mathbf{D}$. Formally, the following three statements are all true:

$$\begin{array}{lll} \mathbf{P}, \mathbf{D}, \mathbf{D} + \{q(b)\} & \models & \exists X\,[p(X) \otimes ins{:}q(X)] \\ \mathbf{P}, \mathbf{D}, \mathbf{D} + \{q(c)\} & \models & \exists X\,[p(X) \otimes ins{:}q(X)] \\ \mathbf{P}, \mathbf{D}, \mathbf{D} + \{q(d)\} & \models & \exists X\,[p(X) \otimes ins{:}q(X)] \end{array}$$

□

2.4 Rules

Rules are formulas of the form $p \leftarrow \phi$, where p is an atomic formula and ϕ is any $\mathcal{TR}$ formula. As in classical logic, this formula is just a convenient abbreviation for the formula $p \vee \neg\phi$. This is the formal, declarative interpretation of rules. In addition, rules in $\mathcal{TR}$ have a procedural interpretation. Intuitively, the formula $p \leftarrow \phi$ means, "to execute p, it is sufficient to execute ϕ." This procedural interpretation is important because it provides $\mathcal{TR}$ with a subroutine facility and makes logic programming possible. For instance, in the rule $p(X) \leftarrow \phi$, the predicate symbol p acts as the name of a procedure, the variable X acts as an input parameter, and the formula ϕ acts as the procedure body, or definition (exactly as in Horn-clause logic programming). Although the rule-body may be any $\mathcal{TR}$ formula, it will frequently be a serial conjunction. In this case, the rule has the form $a_0 \leftarrow a_1 \otimes a_2 \otimes ... \otimes a_n$, where each a_i is an atomic formula. With such rules, users can define transaction subroutines and write transaction logic programs. Note that this facility is possible because transactions are represented by predicates. This property distinguishes $\mathcal{TR}$ from other logics of action, especially those in which actions are modal operators. In such logics, subroutines are awkward, if not impossible, to express. Finally, for notational convenience, we assume that all free variables in a rule are universally quantified outside the rule. Thus, the rule $p(X) \leftarrow \phi$ is simply an abbreviation for $\forall X\,[p(X) \leftarrow \phi]$.

Example 2.9 (A Simple Rule) The rule $c \leftarrow ins{:}a \otimes ins{:}b$ defines a transaction called c. Intuitively, this rule says, "to do c, first insert a into the database, and then insert b." Thus, if the user invokes transaction $?-\ c$, then the database changes from $\mathbf{D}$ to $\mathbf{D} + \{a\}$ to

$\mathbf{D}+\{a,b\}$. Formally, if transaction base $\mathbf{P}$ contains this rule, then the following statement is true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D}+\{a\}, \mathbf{D}+\{a,b\} \quad \models \quad c$$

The simplicity of our examples may have blurred an important point made earlier, namely, that the database itself may be a full-fledged deductive database. For instance, $\mathbf{D}$ may contain the rule $d \longleftarrow a \wedge b$. In this case, the query $?-\ d$ would be answered affirmatively if asked *after* the transaction c has been executed. That is, the following statement would be true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D}+\{a\}, \mathbf{D}+\{a,b\} \quad \models \quad c \otimes d$$

In addition, transactions can modify the rules found in the database. For instance, a transaction may be programmed to delete the formula $d \longleftarrow a \wedge b$ from the database. $\Box$

The rule in Example 2.9 defines a deterministic transaction c. However, rules can also define non-deterministic transactions. To take a simple example, the rule $c \leftarrow \phi \vee \psi$ defines c to be the non-deterministic transaction $\phi \vee \psi$. More generally, rules can define non-deterministic transactions even without mentioning disjunctions explicitly. This is possible because, as in classical logic, a rule like $c \leftarrow (\phi \vee \psi)$ is equivalent to a conjunction of two rules, $(c \leftarrow \phi) \wedge (c \leftarrow \psi)$.

Even a single rule can give rise to non-determinism. This is possible because, as in classical logic, a rule like $\forall X\, [p \leftarrow \phi(X)]$ is equivalent to the rule $p \leftarrow \exists X\, \phi(X)$, and the body of the latter rule is a non-deterministic transaction. Another way to see this is to think of the former rule as a conjunction of its instances, $p \longleftarrow \phi(a)$, for all a. Each of these instances provides an alternate way of executing p, thereby introducing non-determinism.

Example 2.10 (Copying Tuples Non-Deterministically) Example 2.8 shows how one can copy a tuple from one relation to another non-deterministically. We can give this transaction a name by adding the following rule to the transaction base:

$$copy \leftarrow p(X) \otimes ins{:}q(X)$$

Formally, this rule is equivalent to $copy \leftarrow \exists X\, [p(X) \otimes ins{:}q(X)]$. Note that the body of this rule is exactly the formula used in Example 2.8. Intuitively, this rule says, "to execute *copy*, first choose a tuple from relation p non-deterministically; then insert this tuple into relation q." Observe that the set of non-deterministic alternatives is data-dependent. Formally, for every value c,

$$\begin{array}{ll} if & \mathbf{P}, \mathbf{D} \;\models\; p(c) \\ then & \mathbf{P}, \mathbf{D}, \mathbf{D}+\{q(c)\} \;\models\; copy \end{array}$$

This means that we cannot know in advance what the exact outcome of copying will be. However, if the rule above is the only rule defining *copy*, then we can make the following statement, which is true for any pair of databases, $\mathbf{D}_1$ and $\mathbf{D}_2$:

$$
\begin{array}{lrcl}
\textit{if} & \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 & \models & \textit{copy} \\
\textit{then} & \mathbf{P}, \mathbf{D}_1 & \models & p(c) \\
\textit{and} & \mathbf{P}, \mathbf{D}_2 & \models & q(c)
\end{array}
$$

for some value c. □

2.5 Transaction Bases

This section gives simple but realistic examples of transaction bases comprised of finite sets of rules. The examples show how updates can be combined with queries to define complex transactions. In each example, the body of each rule is a sequence of atomic formulas, some of which are queries and some of which are updates.

Example 2.11 (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: $change{:}balance(Acct, Amt1, Amt2)$ 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:

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

In the second and third rules, the atom $balance(Act, Bal)$ is a query that retrieves the balance of the specified account. All other atoms are updates. □

Example 2.12 (Non-Deterministic Robot Actions) The rules below define actions that simulate the movements of a robot arm in the blocks world [14]. States of this world are defined in terms of three database predicates: $on(x, y)$, which says that block x is on top of block y; $clear(x)$, which says that nothing is on top of block x; and $color(x, c)$, which says that c is the color of block x. The rules below define six actions that change the state of the world. Each action evaluates its premises in the order given, and the action fails if any of its premises fail (in which case the database is left in its original state).

$$
\begin{aligned}
&stackSameColor(Z) \leftarrow \\
&\qquad color(Z, C) \otimes stackTwoColors(C, C, Z) \\
&stackTwoColors(C_1, C_2, Z) \leftarrow \\
&\qquad color(X, C_1) \otimes color(Y, C_2) \otimes stackTwoBlocks(X, Y, Z) \\
&stackTwoBlocks(X, Y, Z) \leftarrow \\
&\qquad move(Y, Z) \otimes move(X, Y) \\
&move(X, Y) \leftarrow \\
&\qquad pickup(X) \otimes putdown(X, Y) \\
&pickup(X) \leftarrow \\
&\qquad clear(X) \otimes on(X, Y) \otimes del{:}on(X, Y) \otimes ins{:}clear(Y) \\
&putdown(X, Y) \leftarrow \\
&\qquad X \neq Y \otimes clear(Y) \otimes ins{:}on(X, Y) \otimes del{:}clear(Y)
\end{aligned}
$$

The basic actions are $pickup(X)$ and $putdown(X, Y)$, meaning, "pick up block X," and "put down block X on top of block Y," respectively. Both are defined in terms of elementary inserts and deletes to database relations. The remaining rules combine simple actions into more complex ones. For instance, $move(X, Y)$ means, "move block X to the top of block Y," and $stackTwoBlocks(X, Y, Z)$ means, "stack blocks X and Y on top of block Z." These actions are deterministic: Each set of argument bindings specifies only one robot action.

In contrast, the two actions $stackTwoColors$ and $stackSameColor$ are *non-deterministic*. For instance, $stackTwoColors(C1, C2, Z)$ means, "stack two blocks, of colors C_1 and C_2, on top of block Z." The action does not say which two blocks to use, only their colors. To perform the action, the inference system searches the database for blocks of the appropriate color that can be stacked. If several such blocks are available, the system chooses any two non-deterministically. The action $stackSameColor(Z)$ means, "stack two blocks on top of Z that are the same color as Z." Again, the inference system searches the database for appropriate blocks. In this way, by defining non-deterministic actions, a user can specify what to do (declarative knowledge) as well as how to do it (procedural knowledge). □

It is worth noting that the rules in Example 2.12 involve queries as well as updates. In the last rule, for instance, the atom $clear(Y)$ (which itself may be defined by other deductive rules) is a Boolean test that must return *true* in order for the transaction to continue. In the first rule, the atom $color(Z, C)$ is a query that retrieves the color C of the block Z. The second rule is, perhaps, the most interesting. Here, the atoms $color(X, C_1)$ and $color(Y, C_2)$ are non-deterministic queries. They retrieve two blocks, X and Y, of colors C_1 and C_2, respectively. The particular blocks retrieved by these queries then determine the course of action taken in the rest of the transaction.

Example 2.13 (Recursive, Non-Deterministic Actions) Suppose the transaction base of Example 2.12 is augmented by the two rules below. Then the transaction $stack(N, X)$ tries to stacks N blocks on top of X. If it succeeds, the database is updated and the transaction returns *true*; if it fails (perhaps because N blocks are not available), then the database remains in its original state, and the transaction returns *false*.

$$\begin{aligned} stack(N,X) &\leftarrow N > 0 \otimes move(Y,X) \otimes stack(N-1,Y) \\ stack(0,X) &\leftarrow \end{aligned}$$

In order to stack N blocks on X, the first rule moves a single block, Y, onto X, and then recursively tries to stack $N-1$ blocks on Y. Since the second rule has an empty premise, it terminates the recursion by doing nothing, thereby stacking no blocks on top of X. □

2.6 Constraints

Section 2.3 showed how classical disjunction creates non-determinism. Now we show how classical conjunction constrains it.

In general, the transaction $\psi \wedge \phi$ is more deterministic than either ϕ or ψ by themselves. This is because any execution of $\psi \wedge \phi$ must be an allowed execution of ψ and an allowed execution of ϕ. We illustrate this idea in two ways: first, through an informal example of robot navigation (i.e., a routing problem), and then through two formal and abstract examples. More elaborate and formal examples appear in [4].

Consider the following conjunction of two robot actions:

"Go to the kitchen" $\wedge$ "Don't pass through the bedroom"

Note that each conjunct is a non-deterministic action, since there are many ways in which it can be carried out. In $\mathcal{TR}$, this conjunction would be equivalent to the following:

"Go to the kitchen without passing through the bedroom."

This action is more constrained than either of the two original conjuncts by themselves. In this way, conjunction reduces non-determinism and can specify what is *not* to be done.

Two points about classical conjunction are worth noting. First, it does not cause the conjuncts to be executed as two separate transactions. Instead, it combines them into a single, more tightly constrained transaction. Second, classical conjunction constrains the *entire execution* of a transaction (*i.e.*, the way in which it is carried out), not just the final state of the transaction. It can therefore express *dynamic* integrity constraints.

In general, classical conjunction constrains transactions in two ways: (*i*) by forcing non-deterministic transactions to execute in certain ways, and (*ii*) by causing transactions to fail. The following examples illustrate these points.

Example 2.14 (Transaction Failure: I) If a user issued the transaction $?-\ ins{:}b \wedge del{:}b$, then the transaction would fail,[5] leaving the database unchanged. The transaction fails because it is not possible to simultaneously insert and delete one and the same atom into a database. Formally, the following statements are both true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{b\} \models ins{:}b \qquad \mathbf{P}, \mathbf{D}, \mathbf{D} - \{b\} \models del{:}b$$

[5] We use the term "failure" in the same sense as it is used in Prolog, which is not quite the same as in transaction management theory.

but the following statement is false, for every database $\mathbf{D}'$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D}' \quad \models \quad \mathit{ins{:}b} \wedge \mathit{del{:}b}$$

$\square$

The transaction in Example 2.14 fails because its conjuncts, *ins:b* and *del:b*, terminate at different database states. However, a conjunction $\psi \wedge \phi$ may fail even though the component-transactions, ψ and ϕ, terminate at the *same* state. This is possible because the execution paths of ψ and ϕ may pass through *different* intermediate states, and there may be no execution path that is common to ϕ and ψ.

Example 2.15 (Transaction Failure: II) Each of the transactions, $\mathit{ins{:}b} \otimes \mathit{ins{:}c}$ and $\mathit{ins{:}c} \otimes \mathit{ins{:}b}$, transforms a database, $\mathbf{D}$, into the database $\mathbf{D} + \{b, c\}$. However, they pass through different intermediate states. The former passes through state $\mathbf{D} + \{b\}$, while the latter passes through state $\mathbf{D} + \{c\}$. The conjunction $\mathit{ins{:}b} \otimes \mathit{ins{:}c} \wedge \mathit{ins{:}c} \otimes \mathit{ins{:}b}$ therefore fails, since there is no single sequence of states that is a valid execution path of both conjuncts. Formally, the following two statements are both true:

$$\mathbf{P}, \mathbf{D},\ \mathbf{D} + \{b\},\ \mathbf{D} + \{b, c\} \quad \models \quad \mathit{ins{:}b} \otimes \mathit{ins{:}c}$$
$$\mathbf{P}, \mathbf{D},\ \mathbf{D} + \{c\},\ \mathbf{D} + \{b, c\} \quad \models \quad \mathit{ins{:}c} \otimes \mathit{ins{:}b}$$

Nevertheless, the following statement is false for any database sequence:[6]

$$\mathbf{P}, \mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n \quad \models \quad \mathit{ins{:}b} \otimes \mathit{ins{:}c}\ \wedge\ \mathit{ins{:}c} \otimes \mathit{ins{:}b}$$

$\square$

Example 2.16 (Reducing Non-Determinism) The transactions $\mathit{ins{:}a} \vee \mathit{ins{:}c}$ and $\mathit{ins{:}a} \vee \mathit{ins{:}b}$ are both non-deterministic. However, starting from database $\mathbf{D}$, both transactions can follow the same path and terminate at $\mathbf{D} + \{a\}$. In fact, this is the only database that can be reached by both transactions.[6] Hence, if the user issued the transaction $(\mathit{ins{:}a} \vee \mathit{ins{:}b}) \wedge (\mathit{ins{:}a} \vee \mathit{ins{:}c})$, then after execution, the final database would be $\mathbf{D} + \{a\}$. Formally, the following statement is true:[6]

$$\mathbf{P}, \mathbf{D}, \mathbf{D}' \quad \models \quad (\mathit{ins{:}a} \vee \mathit{ins{:}b})\ \wedge\ (\mathit{ins{:}a} \vee \mathit{ins{:}c}) \qquad \mathit{iff} \qquad \mathbf{D}' = \mathbf{D} + \{a\}$$

In this way, classical conjunction reduces non-determinism and, in this particular example, yields a completely deterministic transaction. $\square$

In [4], we show that $\mathcal{TR}$ is a rich language for expressing constraints. Much of this richness comes from serial conjunction, especially when combined with negation. For example, each of the following formulas has a natural meaning as a constraint:

[6] Assuming b and c are not implied by $\mathbf{D}$.

- $\neg(a \otimes b \otimes c)$ means that the sequence $a \otimes b \otimes c$ is not allowed.
- $\phi \otimes \neg\psi$ means that transaction ψ must *not* follow transaction ϕ.
- $\neg(\phi \otimes \neg\psi)$ means that transaction ψ *must* follow transaction ϕ.

These formulas can often be simplified by using the dual operator $\oplus$, which we call *serial disjunction*. For example, the last formula can be rewritten as $\neg\phi \oplus \psi$ [4].

3 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 the formulas that most users will work with, using them to define transactions and formulate queries. Transaction formulas extend first-order formulas with two new connectives, $\otimes$ and $\oplus$, called *serial conjunction* and *serial disjunction*, respectively.

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 $p \in \mathcal{P}$ is a predicate symbol, and $t_1, \ldots, t_n$ are terms. If ϕ and ψ are transaction formulas, then so are $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \oplus \psi$, $\phi \otimes \psi$, $\neg\phi$, $(\forall X)\phi$, and $(\exists X)\phi$, where X is a variable.

The following are examples of transaction formulas:

- $b(X) \otimes c(X, Y)$
- $a(X) \vee \neg[b(X) \otimes c(X, Y)]$
- $\forall X[a(X) \vee \neg b(X) \oplus \neg c(X, Y)]$

Intuitively, the formula $\psi \otimes \phi$ means, "Do ψ and then do ϕ," and the formula $\psi \oplus \phi$ means, "Do ψ now or do ϕ later." Formally, $\oplus$ and $\otimes$ are dual, so that $\neg(\psi \oplus \phi)$ is equivalent to $\neg\phi \otimes \neg\psi$. Note that classical first-order formulas are transaction formulas that do not use $\otimes$ and $\oplus$.

Transaction formulas combine simple transactions into complex 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 [11, 13, 5]. The 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 a revamping of the entire theory, including the need to reprove soundness and completeness results. In other words, such theories are not extensible. This is a serious problem, since the ability to add new transitions is by no means an esoteric whim. For instance, a transition called *relational assignment* [4] is especially important to SQL-style bulk updates. Furthermore, Katsuno and Mendelzon [9] pointed out that, generally, state transitions belong to two major categories—*updates* and *revisions*—and even within each category several different flavours of such transitions are worth looking at. Thus, there appears to be no small, single set of elementary transitions that is best for all purposes.

For this reason, rather than commit $\mathcal{TR}$ 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. Unlike the axioms in most logics, however, elementary transitions have a special status and are expressed in a special language.

Elementary transitions are specified by formulas of the form $\langle \phi, \psi \rangle u$, where ϕ, ψ are closed first-order formulas and u is a ground (*i.e.*, variable-free) atomic formula, called the *name* of the transition.[7] Intuitively, this formula says that u is an update that transforms database ϕ into database ψ. For instance, in Section 2, the predicates *ins:b* and *del:b* would be defined by an enumerable set of elementary transitions consisting of the formulas $\langle \mathbf{D}, \mathbf{D} + \{b\} \rangle$ *ins:b* and $\langle \mathbf{D}, \mathbf{D} - \{b\} \rangle$ *del:b* for every relational database $\mathbf{D}$. (If $\mathbf{D}$ is a general first-order formula, defining insertion and deletion is more involved; see [9]). Enumerable sets of elementary transitions are called *transition bases.*

The reader is referred to [4] for a more detailed discussion of elementary updates.

4 Model Theory

Just as the syntax is based on two basic ideas—serial conjunction and elementary transitions—the semantics is also based on a small number of fundamental ideas:

- *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 is tantamount to truth over a sequence of database 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 *ins:a* $\otimes$ *ins:b* $\otimes$ *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 unlike modal semantic structures, truth in $\mathcal{TR}$ structures does not hinge on a set of arcs between states. Indeed, $\mathcal{TR}$ structures do not even come with a

[7] Groundedness is imposed for simplicity and because non-ground atoms do not provide new functionality. There is no serious technical problem in handling non-ground atoms in elementary transitions.

set of arcs. Instead, in order to model transactions, we focus on *paths*, that is, on sequences of states. Any finite sequence of states is a valid path. Because of the emphasis on paths, we refer to semantic structures in $\mathcal{TR}$ as *path structures.*

Unlike modal structures, 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. On the other hand, the path $\langle \mathbf{D},\ \mathbf{D}+\{b\},\ \mathbf{D}+\{a,b\} \rangle$ does *not* satisfy this formula. Instead, it satisfies the formula $ins{:}b \otimes ins{:}a$. We carry the idea of truth on paths to its logical conclusion. Thus, if a path satisfies the formulas ψ and $\neg\psi \vee \phi$, then the path must also satisfy the formula ϕ. Note that a path of length 1 corresponds to a single database state. In this way, $\mathcal{TR}$'s path-based semantics accounts for databases as well as transactions. That is, with just one model-theoretic device, paths, we can give meaning to statements about databases and to statements about transactions.

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, which may contain quite complex first-order formulas. 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 state, s, thus corresponds to a particular database—the database having precisely those models that comprise s.

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 clear, 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 consensus on the answer to this question, and many solutions have been proposed (see [9] for a comprehensive discussion). Similar difficulties exist for deletion. Furthermore, insertions and deletions are not always the best choice for elementary updates, as mentioned earlier.

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.[8] Elementary updates are defined axiomatically by a transition base, as outlined in Section 3.

Executional Entailment: Executional entailment is a statement of the form

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi \tag{2}$$

where $\mathcal{B}$ is a transition base, $\mathbf{P}$ is a transaction base, ϕ is a transaction formula, and $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ is a sequence of databases (first-order formulas).

[8] This is only one source of non-determinism is $\mathcal{TR}$, since, as shown earlier, non-determinism may arise even when all elementary updates are deterministic.

Informally, (2) is true if transaction ϕ, defined by the transaction base $\mathbf{P}$, is true on the path $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$. This idea ties together the model theory of $\mathcal{TR}$ and the intuitive concept of execution: if (2) is true and the user has issued a transaction $?- \ \phi$, this results in an execution of this transaction along some path of database states. The path $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ is one such possible path and $\mathbf{D}_n$ is one of the possible final states of the execution. This concept was illustrated in Section 2 and its formal rendering can be found in [4].

5 Proof Theory

$\mathcal{TR}$ has a sound and complete proof theory. Furthermore, there is a subset of $\mathcal{TR}$, which we call *Horn* $\mathcal{TR}$, that corresponds to classical logic programs and that has an efficient SLD-style proof procedure. Although we cannot give the proof theory adequate coverage in this survey, the following properties of executional entailment form the basis of a "naive" proof theory:[9]

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.

We refer the reader to [4] for a thorough treatment of the proof theory.

6 Conclusions

We have reviewed the main ideas underlying Transaction Logic, a novel declarative formalism for specifying and executing actions. A wide range of applications were alluded to, but their full treatment appears elsewhere [4]. These applications include (but are not limited to): view updates in databases, logic programming with "clean" *assert* and *retract*, object-oriented logic programming, temporal constraints among actions, planning robot actions, the frame problem in AI, subjunctive and counterfactual reasoning, program verification, and others.

Acknowledgments: We would like to thank Mariano Consens, Alberto Mendelzon, and Ray Reiter for comments and critique, which inspired many improvements. We are also gratefully acknowledging productive discussions with Gösta Grahne and Peter Revesz.

[9] Here, $\mathcal{B}$, $\mathbf{P}$, and $\mathbf{D}_0, \ldots, \mathbf{D}_n$ are as in (2) above, and α and β are transaction formulas.

References

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

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

[3] A.J. Bonner and M. Kifer. A general logic of state change. Technical report, CSRI, University of Toronto, 1992. In preparation.

[4] A.J. Bonner and M. Kifer. Transaction logic programming. Technical Report CSRI-270, University of Toronto, April 1992. Revised: December 1992. (This report can be FTP'ed (in binary mode) from the anonymous FTP account at cs.sunysb.edu; file: pub/TechReports/kifer/transaction-logic.dvi.Z)

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

[6] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[7] D. Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

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

[9] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge database and revising it. In *Proc. of the 2-nd Intl. Conference on Principles of Knowledge Representation (KR'91)*, pages 387–394, Boston, Mass., April 1991.

[10] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 90/14, Department of Computer Science, SUNY at Stony Brook, July 1990. To appear in J. of ACM.

[11] 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.

[12] J.M. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969. Reprinted in *Readings in Artificial Intelligence*, 1981, Tioga Publ. Co.

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

[14] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publ. Co., Paolo Alto, CA, 1980.

[15] R. Reiter. Formalizing database evolution in the situation calculus. In *Conf. on Fifth Generation Computer Systems*, 1992.