An Overview of Transaction Logic

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

Michael Kifer†
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.
kifer@cs.sunysb.edu

Appears in *Theoretical Computer Science (TCS)*, 133:205–265, 1994.

Abstract

This paper presents an overview of *Transaction Logic*—a new formalism recently introduced in [11, 12] and designed to deal with the phenomenon 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* procedures that accomplish state transitions in a logically sound 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.

This paper is available at the following URL:
`ftp://db.toronto.edu/pub/bonner/papers/transaction.logic/tcs94.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.

Contents

1 Introduction

This paper offers an informal introduction to a novel formalism, called Transaction Logic (abbreviated $\mathcal{TR}$), that is specifically designed to update arbitrary logical theories. 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 provides a tutorial introduction to $\mathcal{TR}$, develops the Herbrand model theory, discusses the issues involved, and compares $\mathcal{TR}$ to related works. The paper also describes other aspects of $\mathcal{TR}$, including hypothetical and retrospective reasoning, negation-as-failure, and updates of non-first-order databases. A general, non-Herbrand semantics and a sound-and-complete proof theory are presented in [11, 12].

$\mathcal{TR}$ 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. An extensive discussion of a number of applications of $\mathcal{TR}$, both practical and theoretical, appears in [11, 14]. 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 [35], to provide a logical account of *methods*, that is, of programs that comprise objects' interfaces and update their internal states. Thus, while F-logic covers the structural aspect of object-oriented databases, its combination with $\mathcal{TR}$ covers 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 in specifying transactions than procedural systems. 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, users may spell out an exact sequence of operations in excruciating detail. At the other extreme, they may specify loose constraints that the transaction must satisfy. In general, sequences of actions and constraints can be arbitrarily mixed, and in this way, procedural and declarative knowledge are seamlessly integrated.

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

[1] STRIPS was an early AI planning system that simulated the actions of a robot arm [22].

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 Cobol. 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. This issue receives extensive treatment in [11].

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 [4, 3, 51]. 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) insist on strict separation of 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 [41, 42], actions are function terms, while in Dynamic and Process Logics [32, 33], 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 [33], 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.

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 $\mathcal{TR}$'s model theory; it is described in detail in [11].

2 Overview and Introductory Examples

Using $\mathcal{TR}$ from the user's point of view 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 should be mentioned at the outset: *non-deterministic* transactions. 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 tell (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, 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 indeterminate, *i.e.*, it cannot be predicted at the outset, 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, because 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 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 represented as finite sets of formulas; however, they are classical first-order formulas. These formulas are not restricted to be ground atomic facts, though, 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 the 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 the user's perspective, a $\mathcal{TR}$ logic program has two parts: a transaction base, $\mathbf{P}$, which the programmer provides,[2] 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 connectives, such as $\wedge$, $\vee$ and $\neg$, and a new connective 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 restricted neither to this type of databases, nor to these particular elementary update operators.

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. Typically, these definitions are constructed from formulas that are akin to the Horn rules of classical logic programming. We call these formulas *serial-Horn* rules. These rules 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 inference in $\mathcal{TR}$ reduces to first-order inference. 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 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 $lucky$, and that the transaction base, $\mathbf{P}$, contains the rule $happy \leftarrow lucky$. By typing $?-\ happy$, the user is posing a query, asking the $\mathcal{TR}$ interpreter if $happy$ is true. In this case, $happy$ can be inferred from the contents of the database and the transaction base, so the interpreter returns "true." Likewise, the interpreter returns "true" if the user types $?-\ lucky$ or $?-\ lucky \wedge happy$. Note that in each case, the database remains unchanged. We represent this behavior formally by the following three statements:

$$\mathbf{P}, \mathbf{D} \models lucky \qquad \mathbf{P}, \mathbf{D} \models happy \qquad \mathbf{P}, \mathbf{D} \models lucky \wedge happy \qquad \Box$$

[2]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.

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 call the simplest kind of updates *elementary updates* or *elementary state transitions*. In principle, there are no restrictions 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 paper, we use the insertion and deletion of atomic formulas as canonical examples of elementary updates. However, many other kinds of elementary updates are also useful. In [11, 14], we discuss several of them, including what we call *bulk updates* and *non-deterministic sampling*.

Elementary updates are atomic in that they cannot be decomposed into simpler updates. We therefore represent them by atomic formulas. Like all formulas, elementary updates can have *both* a truth value *and* a side effect on the database. We represent this idea formally by executional entailments 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, in this paper, for each predicate symbol p, we 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) Suppose *in* is a binary predicate symbol. Then the atoms $ins{:}in(pie, sky)$ and $del{:}in(pie, sky)$ are elementary updates. Intuitively, $ins{:}in(pie, sky)$ means, "insert the atom $in(pie, sky)$ into the database." Likewise, the atom $del{:}in(pie, sky)$ means, "delete $in(pie, sky)$ from the database." From the user's perspective, typing $?-\ ins{:}in(pie, sky)$ to the interpreter changes the database from $\mathbf{D}$ to $\mathbf{D} + \{in(pie, sky)\}$. Likewise, typing $?-\ del{:}in(pie, sky)$ changes the database from $\mathbf{D}$ to $\mathbf{D} - \{in(pie, sky)\}$. We express this behaviour formally by the following two statements, which are true for any transaction base $\mathbf{P}$:

$$\begin{aligned} \mathbf{P}, \mathbf{D}, \mathbf{D} + \{in(pie, sky)\} &\models ins{:}in(pie, sky) \\ \mathbf{P}, \mathbf{D}, \mathbf{D} - \{in(pie, sky)\} &\models del{:}in(pie, sky) \end{aligned} \qquad \Box$$

Here we use the operator "+" to denote set union when it is applied to databases. This is possible because in most of our examples, a database is a set 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 an arbitrary first-order formula, say, a logic program.

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 $\mathcal{TR}$ and the connection between them is established via axioms of the so called *transition base*, as explained later (Section 3.2).

$\mathcal{TR}$ 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 facilities provided in $\mathcal{TR}$.

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.[3] 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:poor* $\otimes$ *ins:sad*, where *poor* and *sad* are ground atomic formulas, denotes a sequence of two insertions. This transaction means, "First insert *poor* into the database, and then insert *sad*." Thus, if the initial database is $\mathbf{D}$, and if the user issues a transaction by typing $?-$ *ins:poor* $\otimes$ *ins:sad*, then during execution, the database will change from $\mathbf{D}$ to $\mathbf{D} + \{poor\}$ to $\mathbf{D} + \{poor, sad\}$. We express this behaviour formally by the following statement, which is true for any transaction base $\mathbf{P}$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{poor\}, \mathbf{D} + \{poor, sad\} \models \mathit{ins{:}poor} \otimes \mathit{ins{:}sad} \qquad \Box$$

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

[3] 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 $\neg\psi \oplus \neg\phi$.

$$\begin{array}{llcl} \textit{if} & \mathbf{P}, \mathbf{D} & \models & \textit{short} \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{sweet\} & \models & \textit{short} \otimes \textit{ins:sweet} \end{array} \qquad \Box$$

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 eliminating 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 [42, 55].

Example 2.5 (Post-Conditions) If the atom *happy* is a query (*i.e.*, has no side effects), then the expression $\textit{ins:won} \otimes \textit{happy}$ denotes an update followed by a test. Intuitively, it means, "insert *won* into the database, and then check that *happy* is true." In other words, the database state is first changed from $\mathbf{D}$ to $\mathbf{D} + \{won\}$. If *happy* 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,

$$\begin{array}{llcl} \textit{if} & \mathbf{P}, \mathbf{D} + \{won\} & \models & \textit{happy} \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{won\} & \models & \textit{ins:won} \otimes \textit{happy} \end{array} \qquad \Box$$

2.3 Non-Deterministic Transactions

In addition to sequencing, 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) Suppose the transaction is $?-\ \textit{ins:won} \vee \textit{ins:lost}$. Then, after execution, the final database will be either $\mathbf{D} + \{won\}$ or $\mathbf{D} + \{lost\}$. In effect, this states that the user does not care (or has no control over) which update will be made, so the system chooses one of them and executes it. Formally, both of the following statements are true, for any transaction base $\mathbf{P}$:

$$\begin{array}{lcl} \mathbf{P}, \mathbf{D}, \mathbf{D} + \{won\} & \models & \textit{ins:won} \vee \textit{ins:lost} \\ \mathbf{P}, \mathbf{D}, \mathbf{D} + \{lost\} & \models & \textit{ins:won} \vee \textit{ins:lost} \end{array} \qquad \Box$$

In addition to defining non-deterministic transactions, it is worth noting that 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 classical disjunction is used to define transactions, the non-deterministic 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 $?-\ ins{:}has(X)$ (or, equivalently, $?-\ \exists X\ ins{:}has(X)$). The database system will then insert a tuple, $\langle c\rangle$, into relation has for some non-deterministically chosen value c. That is, after execution, the database state will be $\mathbf{D}+\{has(c)\}$. Formally, the following statement is true for every value c:

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

□

Example 2.8 (Non-Deterministic Copy) Consider $?-\ \exists X\,[handsome(X) \otimes ins{:}hired(X)]$, a transaction that requests the system to copy a tuple from *handsome* to *hired* non-deterministically. That is, the system will first choose some arbitrary tuple, $\langle mary\rangle$, from relation *handsome*, and then insert the tuple into relation *hired*. Formally, for every value c,

$$\begin{array}{ll} \textit{if} & \mathbf{P}, \mathbf{D} \models handsome(c) \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D}+\{hired(c)\} \quad \models \quad \exists X\,[handsome(X) \otimes ins{:}hired(X)] \end{array}$$

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

$$\begin{array}{lcl} \mathbf{P}, \mathbf{D}, \mathbf{D}+\{hired(mary)\} & \models & \exists X\,[handsome(X) \otimes ins{:}hired(X)] \\ \mathbf{P}, \mathbf{D}, \mathbf{D}+\{hired(bill)\} & \models & \exists X\,[handsome(X) \otimes ins{:}hired(X)] \\ \mathbf{P}, \mathbf{D}, \mathbf{D}+\{hired(kate)\} & \models & \exists X\,[handsome(X) \otimes ins{:}hired(X)] \end{array}$$

Note that even though the above transaction can be executed in several ways (with different values for X), only *one* value will be chosen in any concrete execution, and so only one person will be hired. □

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 $a_0 \leftarrow a_1 \otimes a_2 \otimes \ldots \otimes a_n$, 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 $win \leftarrow ins{:}divided \otimes ins{:}conquered$ defines a transaction called win. Intuitively, this rule says, "to do win, first insert $divided$ into the database, and then insert $conquered$." Thus, if the user invokes transaction $?-\ win$, then the database changes from $\mathbf{D}$ to $\mathbf{D}+\{divided\}$ to $\mathbf{D}+\{divided, conquered\}$. Formally, if transaction base $\mathbf{P}$ contains this rule, then the following statement is true:

$$\mathbf{P}, \mathbf{D},\ \mathbf{D}+\{divided\},\ \mathbf{D}+\{divided, conquered\} \quad \models \quad win$$

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 $unhappy \longleftarrow divided \wedge conquered$. In this case, the query $?-\ unhappy$ will be answered affirmatively if asked *after* the transaction win has been executed. That is, the following statement is true:

$$\mathbf{P}, \mathbf{D},\ \mathbf{D}+\{divided\},\ \mathbf{D}+\{divided, conquered\} \quad \models \quad win \otimes unhappy$$

In addition, as explained in Section 7, transactions can modify the rules found in the database. For instance, a transaction may be programmed to delete the formula $unhappy \longleftarrow divided \wedge conquered$ from the database. $\Box$

The rule in Example 2.9 defines a deterministic transaction. 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 the two rules, $(c \leftarrow \phi) \wedge (c \leftarrow \psi)$.

Example 2.10 (Flipping Coins) Suppose the transaction base $\mathbf{P}$ contains the following two rules, which define the action of flipping a coin:

$$flip(X) \leftarrow ins{:}heads(X) \qquad\qquad flip(X) \leftarrow ins{:}tails(X)$$

These rules say that, to flip a coin, say $dime$, either insert $heads(dime)$ into the database or insert $tails(dime)$. Thus, $?-\ flip(dime)$ is a non-deterministic transaction. Formally, the result of flipping a dime can be represented by the following two statements:

$$\mathbf{P}, \mathbf{D},\ \mathbf{D}+\{heads(dime)\} \quad \models \quad flip(dime) \qquad\qquad \mathbf{P}, \mathbf{D},\ \mathbf{D}+\{tails(dime)\} \quad \models \quad flip(dime)$$

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

$$\begin{array}{lll} \textit{if} & \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 & \models \quad flip(dime) \\ \textit{then} & \mathbf{P}, \mathbf{D}_2 & \models \quad heads(dime) \vee tails(dime) \end{array}$$

In other words, regardless of the outcome, the coin will always land on one of its two sides. $\Box$

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 2.11 (Non-Deterministic Copy, Continued) In Example 2.8 we showed that it is possible to non-deterministically copy a tuple from one relation to another. We can give this transaction a name by adding the following rule to the transaction base, $\mathbf{P}$:

$$hire \leftarrow handsome(X) \otimes ins{:}hired(X)$$

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

$$\begin{array}{lll} \textit{if} & \mathbf{P}, \mathbf{D} & \models \; handsome(c) \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{hired(c)\} & \models \; hire \end{array}$$

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

$$\begin{array}{lll} \textit{if} & \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 & \models \; hire \\ \textit{then} & \mathbf{P}, \mathbf{D}_1 & \models \; handsome(c) \\ \textit{and} & \mathbf{P}, \mathbf{D}_2 & \models \; hired(c) \end{array}$$

for some value c. $\square$

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, and they show how $\mathcal{TR}$ improves upon Prolog's update operators. 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. Each such atom specifies a transaction subroutine (or sub-transaction).

Example 2.12 (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. In the *transfer* transaction, we also require that the debited account should not to be overdrawn. These transactions are defined by the following four rules:

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

Note that, in these rules, the atom $balance(Acct, Bal)$ is a query that retrieves the balance of the specified account, while $Bal \geq Amt$ is a test. All other atoms are updates. □

Observe that the rules in Example 2.12 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:

$$?- \ transfer(fee, client, broker) \otimes transfer(cost, client, seller)$$

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 a 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.

The next example uses robot actions to illustrate non-deterministic rules. Planning of robot actions is discussed in detail in [11].

Example 2.13 (Non-Deterministic Robot Actions) The rules below define actions that simulate the movements of a robot arm in the blocks world [50]. States of this world are defined in terms of four database predicates: $on(x, y)$, which says that block x is on top of block y; $wider(x, y)$, which says that x is wider than y; $isclear(x)$, 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 fails (in which case the database is left in its original state).

$$
\begin{array}{l}
stackSameColor(Z) \leftarrow color(Z, C) \otimes stackTwoColors(C, C, Z). \\
stackTwoColors(C_1, C_2, Z) \leftarrow color(X, C_1) \otimes color(Y, C_2) \otimes stackTwoBlocks(X, Y, Z) \\
stackTwoBlocks(X, Y, Z) \leftarrow move(Y, Z) \otimes move(X, Y) \\
move(X, Y) \leftarrow pickup(X) \otimes putdown(X, Y) \\
pickup(X) \leftarrow isclear(X) \otimes on(X, Y) \otimes del{:}on(X, Y) \otimes ins{:}isclear(Y) \\
putdown(X, Y) \leftarrow X \neq Y \otimes wider(Y, X) \otimes isclear(Y) \otimes ins{:}on(X, Y) \otimes del{:}isclear(Y)
\end{array}
$$

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." Similarly, $putdown(X, Y)$ means that block X

is to be stacked on top of Y, provided that X and Y are not the same and that Y is a wider block. 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 arbitrarily. The action $stackSameColor(Z)$ means, "stack two blocks on top of Z that are of 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) and how to do it (procedural knowledge). □

It is worth noting that the rules in Example 2.13 involve queries as well as updates. In the last rule, for instance, the atom $isclear(Y)$ (which itself may be defined by other deductive rules) is a Boolean test that must return *true* in order for the transaction to succeed. 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.

Just as in Example 2.12, the above $\mathcal{TR}$ program can be straightforwardly converted into a Prolog program and, just as before, this program will not work as intended. Indeed, suppose the robot had picked up a block, $blkA$, that is *wider* than any *clear* block currently on the table. Because *putdown* checks that a wider block does not go on top of a smaller one, it will fail. Once again, in $\mathcal{TR}$, the *putdown* action will fail and the changes to the underlying database will be undone. But not so in Prolog, which will leave the robot stranded in an inconsistent internal state.

Example 2.14 (Recursive, Non-Deterministic Actions) Suppose the transaction base of Example 2.13 is augmented by the two rules below. Then the transaction $stack(N, X)$ tries to stack 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. This section shows 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 examples. More elaborate and formal examples are given in [11].

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 alone. 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 (see [11] for more details).

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

Example 2.15 (Transaction Failure: I) If a user issued the transaction $?- \; ins{:}hired \wedge del{:}hired$, then the transaction would fail,[4] 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:

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

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

$$\mathbf{P}, \mathbf{D}, \mathbf{D}' \;\models\; ins{:}hired \wedge del{:}hired$$

□

The transaction in Example 2.15 fails because its two conjuncts, *ins:hired* and *del:hired*, 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 path of ψ and ϕ may pass through *different* intermediate states and there is no execution path common to ϕ and ψ.

Example 2.16 (Transaction Failure: II) Consider a pair of transactions $?- \; ins{:}bought \otimes ins{:}wanted$ and $?- \; ins{:}wanted \otimes ins{:}bought$. They both transform a database state, $\mathbf{D}$, into the state $\mathbf{D} + \{bought, wanted\}$. However, they pass through different intermediate states. The former goes through the state $\mathbf{D} + \{bought\}$, while the latter passes through the state $\mathbf{D} + \{wanted\}$. The conjunction $(ins{:}bought \otimes ins{:}wanted) \wedge (ins{:}wanted \otimes ins{:}bought)$ 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:

$$\begin{aligned} \mathbf{P}, \mathbf{D}, \mathbf{D} + \{bought\}, \mathbf{D} + \{bought, wanted\} &\;\models\; ins{:}bought \otimes ins{:}wanted \\ \mathbf{P}, \mathbf{D}, \mathbf{D} + \{wanted\}, \mathbf{D} + \{bought, wanted\} &\;\models\; ins{:}wanted \otimes ins{:}bought \end{aligned}$$

[4] 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 any sequence of databases $\mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n$:[5]

$$\mathbf{P}, \mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (ins{:}bought \otimes ins{:}wanted) \land (ins{:}wanted \otimes ins{:}bought) \qquad \Box$$

Example 2.17 (Reducing Non-Determinism) Consider the two transactions $?-\ ins{:}lost \lor ins{:}found$ and $?-\ ins{:}lost \lor ins{:}won$. They are both non-deterministic. Furthermore, starting from database $\mathbf{D}$, they can both follow the same path to terminate at $\mathbf{D} + \{lost\}$. In fact, this is the only database that can be reached by both transactions.[6] Hence, if the user issued the transaction $?-\ (ins{:}lost \lor ins{:}found) \land (ins{:}lost \lor ins{:}won)$ then, after execution, the final database would be $\mathbf{D} + \{lost\}$. Formally, the following statement is true:[6]

$$\mathbf{P}, \mathbf{D}, \mathbf{D}' \models (ins{:}lost \lor ins{:}found) \land (ins{:}lost \lor ins{:}won) \quad \textit{iff} \quad \mathbf{D}' = \mathbf{D} + \{lost\}$$

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

In [11], 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:

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

Finally, it is worth noting that in addition to specifying constraints, classical conjunction has other important functions. For instance, as in classical logic, it is needed to specify conjunctive queries. Likewise, as in deductive databases, it is needed to construct rule-bases, which are just conjunctions of rules. Of course, unlike deductive databases, rules in $\mathcal{TR}$ define transactions as well as queries. Examples of such rules are given in the next section.

3 Syntax

The syntax of $\mathcal{TR}$ distinguishes two kinds of formulas: *transaction formulas* and *elementary state transitions*. The former define composite transactions, and the latter define elementary updates. The rest of this section defines these two types of formulas precisely and discusses them in more detail.

We define the alphabet of a language of $\mathcal{TR}$ to consist of the following symbols:

- A set of function symbols $\mathcal{F}$. Each function symbol has a non-negative number, called its *arity*, indicating the number of arguments the symbol can take. Constants are viewed as 0-ary function symbols.

[5] Assuming *bought* and *wanted* are not in $\mathbf{D}$.
[6] Assuming *found* and *won* are not in $\mathbf{D}$.

- A countably-infinite set of variables $\mathcal{V}$.

- A countable set $\mathcal{P}$ of predicate symbols. Like functions, predicate symbols have arity. 0-ary predicate symbols are viewed as propositional constants.

- Logical connectives $\vee$, $\wedge$ (classical disjunction and conjunction), $\oplus$, $\otimes$ (serial disjunction and conjunction), $\neg$ (classical negation). Additional connectives will be defined in terms of these later.

- Quantifiers $\forall$, $\exists$.

- Auxiliary symbols, such as "(", ")", "$\rangle$", "$\langle$", and ",".

Terms are defined as usual in first-order logic: A variable is a term; if f is an n-ary ($n \geq 0$) function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. When $n = 0$, we write f instead of $f(\,)$.

3.1 Transaction Formulas

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. First, 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. Second, if ϕ and ψ are transaction formulas, then so are the following expressions:

- $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \oplus \psi$, $\phi \otimes \psi$, and $\neg\phi$.

- $(\forall X)\phi$ 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 $\psi \oplus \phi$ (very roughly) means, "Do ψ now or do ϕ later." Formally, $\oplus$ and $\otimes$ are defined to be dual, so that $\neg(\psi \oplus \phi)$ is equivalent to $\neg\phi \otimes \neg\psi$. Note that the classical first-order formulas are transaction formulas that do not use $\otimes$ and $\oplus$. As in classical logic, we introduce convenient abbreviations for complex formulas. For instance, $\psi \leftarrow \phi$ is an abbreviation for $\psi \vee \neg\phi$, for any transaction formulas ψ and ϕ.

3.2 Elementary Transitions

Transaction formulas combine simple transactions into complex ones. However, we also need a way to specify *elementary* changes to the database. One way to define such transitions is to build them into the semantics as in [40, 48, 8, 18, 1, 43]. 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. In [11, 14], for instance, a new kind of transition called *relational assignment* is introduced for performing bulk updates and non-deterministic sampling. In scientific databases, an elementary transition may be a sophisticated numerical operation, such as a Fourier transform. Furthermore, Katsuno and Mendelzon [34] 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 state transitions that is best for all purposes.

For this reason, rather than committing $\mathcal{TR}$ to a fixed set of elementary transitions, we have chosen to treat elementary state 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 state 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 state 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 state 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 [34]). We refer to the formulas themselves as *elementary state transitions*, and we call the enumerable set thereof *transition base*.

The names of elementary transitions, such as *ins:b* and *del:b*, have *no* special status in $\mathcal{TR}$. That is, they are ordinary atomic formulas that just happened to be mentioned in the transition base. In principle, nothing prevents the user from putting rules for *ins:b* into the transaction base.[8] Even the fancy names of these predicates is nothing but a convention adopted for this paper.

Although the *names* of elementary transitions are not special, the formulas defining them are. In particular, these formulas *cannot* be combined with transaction formulas via logical connectives.[9] Furthermore, elementary transitions cannot be combined using the new serial connectives of $\mathcal{TR}$, $\oplus$ and $\otimes$, *i.e.*, say, $\langle\psi_1, \phi_1\rangle u \otimes \langle\psi_2, \phi_2\rangle v$ is not a formula. Although it is trivial to extend $\mathcal{TR}$ to allow combining elementary transitions via the classical connectives (*e.g.*, $\langle\psi_1, \phi_1\rangle u \vee \langle\psi_2, \phi_2\rangle v$) and quantifiers, we shall not do this here for we see no apparent practical use for this generality.

Nevertheless, since the names u and v are just ordinary atomic formulas, they can be used to build transaction formulas. For instance, $u \otimes v$ is a transaction formula, as is $u \vee v$. Elementary transitions are therefore not used the way transaction formulas are. Instead, as mentioned above, they are best viewed as a parameter of $\mathcal{TR}$, a parameter that determines the particular theory of elementary updates underlying the logic. In the sequel, the term "formula" will be reserved for transaction formulas only; elementary state transitions will be called just that—"elementary transitions."

A logic program in $\mathcal{TR}$ consists of three separate parts: a transaction base, $\mathbf{P}$, which is a set

[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.

[8] In practice, however, preventing the user from tinkering with the definitions of elementary transitions may be a good software-engineering idea.

[9] But see [13], where a more general logic is described.

of (transaction) formulas; the current database state, $\mathbf{D}$, which is a set of first-order formulas, and which is updatable by the transactions defined in $\mathbf{P}$; and a transition base, $\mathcal{B}$, which defines the set of elementary transitions. In general, the transition base may represent any kind of update or revision discussed in [34]. However, unlike the transaction formulas in $\mathbf{P}$, we do not expect the transition base to be coded by casual users. Although the transition base allows for many different theories of state change, we envision that any logic programming system based on $\mathcal{TR}$ is likely to have a carefully selected repertoire of built-in theories of change and a tightly controlled mechanism for adding new ones. This latter mechanism would not be available to ordinary programmers. For this reason, we assume in this paper that the transition base is fixed.

The practical issue of *how* transition bases are to be generated was partly solved for us by Grahne, Mendelzon, and Winslett [29, 58]. Winslett showed that, in general, the problem of updating propositional formulas is NP-hard. Subsequently, though, Grahne and Mendelzon proved that updating sets of ground atoms with arbitrary propositional formulas can be done in polynomial time. More importantly, this result carries over to deductive databases, in which only the extensional part is updated. In this case, as with relational databases, updates are fast and straightforward.

In scientific databases, elementary transitions may involve complex calculations. For example, a two dimensional array is naturally represented as a ternary relation, $A(i, j, v)$, where v is the value of the i, j^{th} array element. An elementary transition could compute the Fourier transform of this array and store the result in another ternary relation, $B(i, j, v)$. Indeed, scientific packages often supply complex operations, like the Fourier transform and matrix inversion, as elementary operations.

All these issues are orthogonal to our work however. In $\mathcal{TR}$, the transition base is viewed as an infinite collection of axioms, and all that matters is the existence of an algorithm to enumerate this collection. Such enumerations are not unusual—they are implicit in most logical inference systems, since axiom sets are normally infinite.[10] Moreover, the enumeration can be carried out by using built-in procedures to perform elementary updates [11]. The resulting inference system has an efficiency comparable to that of procedural programming languages. Finally, we note that transaction definitions are independent of the contents of the transition base and of the specifics of the enumeration algorithm. This latter point contributes to making $\mathcal{TR}$ a lucid and flexible language for defining transactions.

In closing this section, we note a limitation of transition bases: Although they provide an efficient way of *executing* elementary updates, they provide only limited help with *reasoning* about them. In databases and logic programming, however, execution is by far the more important issue. Reasoning can still be performed in $\mathcal{TR}$—it is just that $\mathcal{TR}$ does not offer specific advantages in this sphere. However, many reasoning problems, such as constraints and planning, can often be reduced to execution [11]. In such cases, $\mathcal{TR}$ does offer decided advantages over other reasoning systems. In planning, for instance, the infamous *frame problem* is not an issue [11].

4 Herbrand-style Semantics

This section develops the Herbrand-style semantics of $\mathcal{TR}$, which is most useful in databases and logic programming. A general, non-Herbrand semantics and a sound-and-complete proof theory are given in [11, 12]. The model theory of $\mathcal{TR}$ is based on a few simple ideas. We first discuss these ideas, and then develop the Herbrand model theory in detail.

[10] Consider common "axioms" such as $\phi \vee \neg\phi$, $[\phi \wedge (\phi \rightarrow \psi)] \rightarrow \psi$, and De Morgan's Laws. These are actually axiom *schemata* representing an infinite number of axioms, since they apply to all formulas ϕ and ψ.

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, 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. As in modal logic, a structure in $\mathcal{TR}$ consists of a set of states. 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 structures, truth in $\mathcal{TR}$'s structures does not hinge on a set of arcs between states. Indeed, $\mathcal{TR}$ structures do not even come with a 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*. This simple example illustrates a general idea in $\mathcal{TR}$: *the truth of a formula on a path corresponds to the common-sense notion of transaction execution.* To define truth on paths in a completely general way, we assign a first-order semantic structure to each path, which specifies those atomic formulas that are true on the path. Satisfaction of complex formulas on a path is determined by the semantic structures assigned to the path and its subpaths. Intuitively, all formulas, atomic or complex, that are true on a path represent actions that take place along the path.

Note that a path of length 1 represents a single database state. In this way, $\mathcal{TR}$'s path-based semantics accounts for databases as well as transactions. Thus, with just one model-theoretic device, paths, we can give meaning to statements about databases, elementary updates, and general transactions. Our reliance on paths in $\mathcal{TR}$ is reminiscent of a version of Process Logic described in [33]. However, there is a vast difference in *how* truth is defined and in *what* formulas actually denote. Further comparison is provided in Section 8.

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, deductive databases, or general logic programs, which may contain quite complex first-order formulas. We therefore take a more general approach by defining a state to be a first-order formula, or rather, an *equivalence class* of first-order formulas.

The need to consider equivalence classes rather than just formulas arises because many formulas may represent essentially the same database state. For instance, we do not normally think of $a \wedge b$

and $b \wedge a$ as being two different databases. For formulas in the same equivalence class, we require, at a minimum, that they be logically equivalent in the classical sense. First-order equivalence is thus the coarsest possible equivalence relation that can be used to define database states. For various specialized theories, however, this relationship may have to be refined. For instance, in logic programming, $p \longleftarrow \neg q$ and $q \longleftarrow \neg p$ are not considered to be equivalent programs, even though they are both classically-equivalent to $p \vee q$. In this case, a suitable equivalence relation would have to make finer distinctions than classical equivalence (see Section 7). If we were to simulate Prolog, with its dependence on the order of clauses and literals, the equivalence relation would be even finer, and each equivalence class would consist of just one formula.

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 simple answer to this question, and many solutions have been proposed (see [34] for a comprehensive discussion). Similar difficulties exist for deletion. Furthermore, insertions and deletions are not always the best choice for elementary updates [11].

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 (actually, an equivalence class of first-order formulas—per the discussion above). More generally, an elementary update may be non-deterministic, so it is not just a mapping, but a *binary relation* on databases.[11] Elementary updates are defined axiomatically by a transition base, as outlined in Section 3.

In sum, a path structure in $\mathcal{TR}$ is a collection of states, where each state is an equivalence class of first-order formulas; a path is a sequence of states (representing transaction execution). We say that the path $\langle s_1, s_2, ..., s_n \rangle$ has length n and that it passes through each of the states s_i. As a special case, the singleton-path $\langle s \rangle$ has length 1 and passes through the state s. Finally, every transaction formula has a truth value (true or false) on every path.

Stored Facts vs. Derived Facts

In classical database systems, a subtle distinction is made between stored facts and derived facts. Typically, update procedures distinguish between these two types of facts, but query processors do not. For example, a deductive database has a set of base facts, $\mathbf{D}$, called the *extensional* database, and a set of rules, $\mathbf{P}$, called the *intensional* database. The query processor, however, sees only the logical consequences of $\mathbf{D} \cup \mathbf{P}$. In this way, a user cannot tell whether a fact is stored or derived. In contrast, update-procedures act only on the stored facts, $\mathbf{D}$. Even if a user requests an update to a derived fact (a view update), the request is translated into an equivalent set of updates to the stored facts. The reason, of course, is that derived facts cannot be changed without changing the base facts that support them.

To reflect this distinction, the path structures of $\mathcal{TR}$ carefully distinguish between a state, s, and the path $\langle s \rangle$ of length 1. The state s represents the formulas stored in the database, and the path $\langle s \rangle$ represents formulas derived by combining the database and the transaction base. That is, in database

[11] This is only one source of non-determinism in $\mathcal{TR}$, since as in Example 2.10, non-determinism may arise even when all elementary updates are deterministic.

terminology, $\langle s \rangle$ is a *view* of s.[12] Only the state s (the stored data) is updated, and only the path $\langle s \rangle$ (the view) is queried. The user, therefore, sees the state indirectly, through the view. To capture this idea formally, formulas are *never* evaluated at states (unlike modal logic). Thus, one cannot ask for the value of a formula, ψ, at a state, s, since this would be tantamount to querying stored facts directly. However, ψ can be evaluated on the path $\langle s \rangle$, which is tantamount to querying the view over s, as in the classical theory of deductive databases.

More generally, *every* user transaction corresponds to evaluating a formula on a *path*. If the path has length 1, then the transaction is a query in the traditional sense; if the path has length 2, then the transaction (usually) is an elementary update; and if the path has length greater than 2, then the transaction is a composite update. In this way, views, updates, and transactions are all treated uniformly.

4.1 Herbrand Path Structures

This section makes the preceding discussion precise.

In Herbrand semantics, the domain and the interpretation of function symbols is fixed, both being determined by the underlying language of the logic. More precisely, given a language of $\mathcal{TR}$, $\mathcal{L}$, with set of function symbols, $\mathcal{F}$ (where we treat constants as 0-ary function symbols), the *Herbrand universe*, denoted $\mathcal{HB}(\mathcal{F})$, is the set of all ground (*i.e.*, variable-free) first-order terms that can be constructed from the symbols in $\mathcal{F}$. Also, recall (from the classical theory of logic programming) that a *Herbrand base* of $\mathcal{L}$ is the set of all ground atomic formulas. A classical *Herbrand structure* of $\mathcal{L}$ is any subset of the Herbrand base.

For our purposes, it is convenient to have one additional, *special,* Herbrand structure, denoted $\top$, which has the property that it satisfies every first-order formula. Even though $\top$ is not a classical structure, here we shall call it "classical" because adding it to classical logic does not change the logic in any essential way. (It simply adds one more model to every formula.) We discuss $\top$ further in Section 4.7.

Finally, we assume that the language $\mathcal{L}$ comes with an equivalence relation, $\overset{\mathcal{L}}{\sim}$, on the collection of all first-order formulas. The only thing that is required of $\overset{\mathcal{L}}{\sim}$ is that it be a *refinement* of the classical first-order equivalence relation, *i.e.*, $\phi \overset{\mathcal{L}}{\sim} \psi$ implies that ϕ and ψ are logically equivalent in the classical sense. A *state* is any equivalence class of first-order formulas with respect to the relation $\overset{\mathcal{L}}{\sim}$. A *path* of length k over $\mathcal{L}$ is an arbitrary finite sequence of states, $\langle s_1, \ldots, s_k \rangle$, where $k \geq 1$. The set of all paths in $\mathcal{L}$ is denoted by $Paths(\mathcal{L})$. In the rest of this paper, the language $\mathcal{L}$ is implicit.

Definition 4.1 (Herbrand Path Structures) A *Herbrand path structure* is a mapping, $\mathbf{M}$, that assigns a classical Herbrand structure to every path; this assigned structure may be the special structure $\top$ introduced earlier. The mapping is subject to the restriction that for every state, s, the semantic structure assigned to the path $\langle s \rangle$ must be a classical model of the formulas in s; that is, $\mathbf{M}(\langle s \rangle) \models^c s$. $\Box$

The mapping $\mathbf{M}$ specifies which atoms are true on what paths. These atoms denote actions that take place along the paths to which they are assigned. As we shall see, these actions can be partially or completely specified by logical formulas. For instance, the formula $a \leftarrow \phi \otimes \psi$ says that action a

[12] Of course, the state s also represents derived information, but only information that is derivable from the formulas in the database.

occurs whenever action ϕ is followed by action ψ. In a logic-programming context, the atom a can be viewed as a *name* for the complex action $\phi \otimes \psi$. The ability to name complex actions is tantamount to having a subroutine facility.

Several points regarding the above definition are worth noting.

- A path structure distinguishes a state, s, from the path $\langle s \rangle$ of length 1. Intuitively, s corresponds to stored data, and $\langle s \rangle$ corresponds to derived data, or views. The requirement that $\mathbf{M}(\langle s \rangle) \models^c s$ formalizes this idea.

- A Herbrand path structure, $\mathbf{M}$, is independent of any particular set of elementary updates. In fact, the definition of path structures does not mention elementary updates at all. This is because elementary updates are defined axiomatically, by logical formulas. These formulas constrain $\mathbf{M}$, forcing certain atoms to be true on paths of length 2 (Section 4.3).

- The Herbrand structure assigned to a path, π, 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 as logical formulas. These formulas constrain $\mathbf{M}$, forcing the atoms on one path to be related in precise ways to the atoms on other paths (Section 4.3).

Because this paper considers only Herbrand path structures, we shall often omit the adjective "Herbrand." Therefore, in the sequel, the term "path structure" should be taken to mean "Herbrand path structure." Furthermore, since states in a path structure are equivalence classes of first-order formulas, we shall often simplify the terminology by referring to formulas as states. Thus, when we say that a formula, $\mathbf{D}$, is a state, we mean that it is a member of the equivalence class that makes up the state. Likewise, if $\mathbf{D}_1, \ldots, \mathbf{D}_n$ are formulas, we will often call $\langle \mathbf{D}_1, ..., \mathbf{D}_n \rangle$ a path, which is a short way of saying that $\langle s_1, \ldots, s_n \rangle$ is a path and $s_1, \ldots, s_n$ are states containing $\mathbf{D}_1, \ldots, \mathbf{D}_n$, respectively.

4.2 Satisfaction on Paths

Transaction formulas are evaluated on paths. That is, they are true or false on paths in a structure. Given a path, $\pi = \langle s_1, ..., s_n \rangle$, it is convenient to define a *split* of π to be any pair of subpaths, π_1 and π_2, such that $\pi_1 = \langle s_1, ..., s_i \rangle$ and $\pi_2 = \langle s_i, ..., s_n \rangle$ for some i $(1 \leq i \leq n)$. In this case, we shall write $\pi = \pi_1 \circ \pi_2$. As in classical logic, in order to define the truth value of quantified formulas and of open formulas, it is convenient to introduce variable assignments. A *variable assignment*, ν, is a mapping $\mathcal{V} \longmapsto \mathcal{HB}(\mathcal{F})$, which 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.*, $\nu(f(t_1, \ldots, t_n)) = f(\nu(t_1), \ldots, \nu(t_n))$.

Definition 4.2 (Satisfaction) Let $\mathbf{M}$ be a Herbrand path structure, let π be a path, and let ν be a variable assignment. Then:

1. **Base Case:**
   $\mathbf{M}, \pi \models_\nu p(t_1, \ldots, t_n)$ if and only if $\mathbf{M}(\pi) \models^c_\nu p(t_1, \ldots, t_n)$, for any atomic formula $p(t_1, \ldots, t_n)$, where "$\models^c_\nu$" denotes classical satisfaction in first-order predicate calculus.

2. **Negation:**
   $\mathbf{M}, \pi \models_\nu \neg\phi$ if and only if it is not the case that $\mathbf{M}, \pi \models_\nu \phi$.

3. **Classical Conjunction:**
   $\mathbf{M}, \pi \models_\nu \phi \wedge \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ and $\mathbf{M}, \pi \models_\nu \psi$.

4. **Classical Disjunction:**
   $\mathbf{M}, \pi \models_\nu \phi \vee \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ or $\mathbf{M}, \pi \models_\nu \psi$.

5. **Serial Conjunction:**
   $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ if and only if $\mathbf{M}, \pi_1 \models_\nu \phi$ and $\mathbf{M}, \pi_2 \models_\nu \psi$ for *some* split $\pi_1 \circ \pi_2$ of path π.

6. **Serial Disjunction:**
   $\mathbf{M}, \pi \models_\nu \phi \oplus \psi$ if and only if $\mathbf{M}, \pi_1 \models_\nu \phi$ or $\mathbf{M}, \pi_2 \models_\nu \psi$ for *every* split $\pi_1 \circ \pi_2$ of path π.

7. **Universal Quantification:**
   $\mathbf{M}, \pi \models_\nu (\forall X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *every* variable assignment μ that agrees with ν everywhere except on X.

8. **Existential Quantification:**
   $\mathbf{M}, \pi \models_\nu (\exists X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *some* variable assignment μ that agrees with ν everywhere except on X. □

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

In Definition 4.2, the base case captures the intuition behind transaction execution along a path: In $\mathcal{TR}$, the truth of $p(t_1, \ldots, t_n)$ on a path π means that transaction p *can* execute along π when invoked with arguments $t_1, \ldots, t_n$. Observe further that the meaning of all "classical" connectives ($\wedge$, $\neg$, etc.) is defined in a classical fashion. For these connectives, truth on a path depends only on the path itself. (This similarity is the main reason for us calling these connectives of $\mathcal{TR}$ "classical.") In contrast, for the non-classical connectives, $\otimes$ and $\oplus$, truth depends not on the path, but on its *subpaths*. Serial disjunction, $\oplus$, in particular, has many interesting applications, such as expressing universal constraints on transaction execution [11]. Also note from the definitions that on paths of length 1, both "$\otimes$" and "$\wedge$" reduce to the usual conjunction of classical predicate logic, while both "$\oplus$" and "$\vee$" reduce to the classical "$\vee$." Thus, the "classical" and the "serial" connectives of $\mathcal{TR}$ extend the usual connectives of the classical logic from states to executions (paths), albeit in different ways.

Besides these general comments, two specific points about serial and classical conjunction are worth mentioning:

- Item 5 defines the satisfaction of $\phi \otimes \psi$ on path π in terms of the *subpaths* of π. Intuitively, such a formula represents a transaction with two subtransactions, ϕ and ψ. Item 5, then, formalizes the idea that a serial transaction succeeds if and only if all of its subtransactions succeed. This point was illustrated in Section 2.5.

- The classical conjunction $\phi \wedge \psi$ in Item 4 does *not* specify concurrent execution of the two transactions, ϕ and ψ. Instead it corresponds to their *constrained execution*. Each of these transactions represents non-deterministic actions that can be executed in many possible ways. Their classical conjunction, then, means, "Execute action ϕ in such a way that ψ is satisfied." Section 2.6 has illustrated this point.

Abbreviations

Before going on, we define a few useful abbreviations. 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 $(\psi \leftarrow \phi) \wedge (\psi \rightarrow \phi)$. By replacing $\vee$ and $\wedge$ with $\otimes$ and $\oplus$ we obtain another interesting pair of connectives: the *left serial implication*, $\psi \Leftarrow \phi$, standing for $\psi \oplus \neg\phi$, and the *right* serial implication, $\phi \Rightarrow \psi$, standing for $\neg\phi \oplus \psi$. As follows from Definition 4.2, in plain English these formulas say the following: "action ϕ must be immediately preceded (resp., followed) by action ψ." Note that unlike "$\leftarrow$" and "$\rightarrow$", the connectives "$\Leftarrow$" and "$\Rightarrow$" are not just syntactic variations of each other, *i.e.*, $\psi \Leftarrow \phi$ is not equivalent to $\phi \Rightarrow \psi$; rather, $\psi \Leftarrow \phi$ is equivalent to $\neg\psi \Rightarrow \neg\phi$. Finally, we introduce two more abbreviations, for iterating the operators $\otimes$ and $\oplus$:

$$\otimes^n \psi \equiv \underbrace{\psi \otimes \cdots \otimes \psi}_{n} \qquad\qquad \oplus^n \psi \equiv \underbrace{\psi \oplus \cdots \oplus \psi}_{n}$$

In addition to these abbreviations, we assume that our language contains a predefined propositional constant, `state`, which is true only on paths of length 1, that is, on database states. Formally, for any path structure $\mathbf{M}$ and path π, it is the case that $\mathbf{M}, \pi \models \texttt{state}$ if and only if π is a path of length 1. Thus, `state` is false on every path having more than one state, even on loops like $\langle s, s \rangle$. (Such loops can arise if, for instance, a transaction inserts an atom into a database when the atom is already there. In such cases, the initial and final states of the database are the same.) Another useful constant is `path`, defined as $\texttt{state} \vee \neg\texttt{state}$, which is true on every path in $\mathbf{M}$. In fact, `path` is equivalent to $\phi \vee \neg\phi$ for any formula ϕ, and thus `path` does not increase the expressive power of the logic. In contrast, `state` cannot be independently defined in $\mathcal{TR}$, and so it *does* increase the expressive power of the logic [11]. Section 5 shows that `state` can be expressed using `path` and a hypothetical operator.

It might seem that `state` gives a special status to paths of length 1, since $\mathcal{TR}$ can now distinguish them from all other paths. We shall soon see, however, that one can use `state` to specify paths of *any* given length. Thus, no particular path length is special.

Observations

Armed with the above definitions, it is easy to verify that the following formulas, analogous to De Morgan's laws, are true on every path of every path structure. We call such formulas *tautologies*.

$$\begin{array}{rcl}
\neg(\phi \oplus \psi) & \leftrightarrow & \neg\phi \otimes \neg\psi \\
\neg(\phi \otimes \psi) & \leftrightarrow & \neg\phi \oplus \neg\psi \\
(\phi \vee \psi) \otimes \eta & \leftrightarrow & (\phi \otimes \eta) \vee (\psi \otimes \eta) \\
(\phi \wedge \psi) \oplus \eta & \leftrightarrow & (\phi \oplus \eta) \wedge (\psi \oplus \eta) \\
(\phi \wedge \psi) \otimes \eta & \rightarrow & (\phi \otimes \eta) \wedge (\psi \otimes \eta) \\
(\phi \vee \psi) \oplus \eta & \leftarrow & (\phi \oplus \eta) \vee (\psi \oplus \eta)
\end{array} \tag{2}$$

The first pair of laws establishes duality between the serial connectives $\otimes$ and $\oplus$. The standard duality between $\vee$ and $\wedge$ also holds. The last pair of implications is unidirectional, which means that $\otimes$, $\oplus$ do not fully distribute through $\wedge$ and $\vee$, respectively. Likewise, $\oplus$ and $\otimes$ do not distribute through each other and $\vee$, $\wedge$ do not distribute through $\oplus$ and $\otimes$. The standard De Morgan's laws involving $\vee$, $\wedge$, $\neg$, and quantifiers *do* hold, however. In particular, this implies that every $\mathcal{TR}$-formula has prenex normal form, but not necessarily conjunctive or disjunctive normal forms.

Here are tautologies involving the propositional constant `state`, which is true only on paths of length 1:

$$\phi \leftrightarrow (\phi \otimes \texttt{state})$$
$$\phi \leftrightarrow (\texttt{state} \otimes \phi)$$
$$\phi \leftrightarrow (\phi \Leftarrow \texttt{state})$$
$$\phi \leftrightarrow (\texttt{state} \Rightarrow \phi)$$

Another interesting observation is that `state` enables the construction of formulas that are true on paths of any given length. To do this, we first observe that the formula $\neg\texttt{state}$ is true on paths of length greater than 1, that is, on paths with at least one arc. The formula $\neg\texttt{state} \otimes \neg\texttt{state}$ is therefore true on paths with at least two arcs. More generally, the formula $\otimes^n \neg\texttt{state}$ is true on paths with at least n arcs. Finally, we observe that the negation of this formula has a particularly simple form:

$$\neg \otimes^n \neg\texttt{state} \quad \equiv \quad \oplus^n \texttt{state}$$

This formula is true on paths with fewer than n arcs, *i.e.*, fewer than $n+1$ states. Thus, their length is at most n. We can now define the proposition `n-path`, which is true on paths of length *exactly* n:

$$\texttt{n-path} \quad \equiv \quad \oplus^n \texttt{state} \wedge \neg \oplus^{n-1} \texttt{state}$$

As a special case, we define the proposition `arc` to be equivalent to `2-path`. That is,

$$\texttt{arc} \quad \equiv \quad (\texttt{state} \oplus \texttt{state}) \wedge \neg\texttt{state}$$

This proposition, which is true on arcs, is analogous to the familiar temporal operator `next`. By iterating `arc`, we obtain an alternate definition of `n-path`, namely, $\texttt{n-path} \equiv \otimes^{n-1}\texttt{arc}$.

4.3 Models

Herbrand path structures assign a first-order Herbrand structure, $\mathbf{M}(\pi)$, to each path, π. For arbitrary path structures, this assignment is completely arbitrary. Intuitively, this represents a situation in which we have no knowledge about actions: each ground atom represents an action, but we do not know on which paths this action takes place. When it exists, knowledge about actions is expressed as logical formulas. In this case, we are not interested in *arbitrary* path structures, but in *models* of the logical formulas. In these models, the Herbrand structures assigned by $\mathbf{M}$ are *not* arbitrary, but satisfy constraints and dependencies that are implicit in the logical formulas. Two kinds of logical formula are involved, elementary transitions and transaction formulas. Elementary transitions constrain the Herbrand structures on specific arcs (paths of length 2). Transaction formulas constrain the Herbrand structures on all paths, and can create dependencies between the Herbrand structures on a path and its subpaths. We deal with elementary transitions first.

Recall that an elementary transition is an expression of the form $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$, where u is a ground atomic formula, and $\mathbf{D}_1$ and $\mathbf{D}_2$ are (finite sets of) first-order formulas. Intuitively, this expression means that update u causes a transition from database $\mathbf{D}_1$ to database $\mathbf{D}_2$. We formalize this idea in the next definition.

Definition 4.3 (Models of Transition Bases) Let $\mathbf{M}$ be a path structure, and let $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ be an elementary state transition. The transition is *satisfied* by $\mathbf{M}$ if and only if $\mathbf{M}, \langle \mathbf{D}_1, \mathbf{D}_2 \rangle \models u$. We say that $\mathbf{M}$ is a *path-model* (or simply a *model*) of a transition base, $\mathcal{B}$, if and only if $\mathbf{M}$ satisfies every elementary state transition in $\mathcal{B}$. □

In this definition, $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$ is a path of length 2, and the statement $\mathbf{M}, \langle \mathbf{D}_1, \mathbf{D}_2 \rangle \models u$ is from Definition 4.2.[13] Since u is atomic, only the first item in Definition 4.2 is applicable. Thus, $\mathbf{M}$ satisfies

[13] We can use $\models$ instead of $\models_\nu$ because u is an elementary transition and is, therefore, ground.

$\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ if and only if $u \in \mathbf{M}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)$, *i.e.*, if and only if the atom u is classically true in the Herbrand structure assigned to path $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$ by $\mathbf{M}$.

Definition 4.3 guarantees that every model of a transition base will have a certain amount of structure. For example, if the formulas $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$, and $\langle \mathbf{D}_2, \mathbf{D}_3 \rangle v$ are in the transition base, then the atom u is true on the path $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$, and the atom v is true on the path $\langle \mathbf{D}_2, \mathbf{D}_3 \rangle$. This bare-bones structure has certain logical consequences. For instance, by Item 5 of Definition 4.2, the formula $u \otimes v$ is true on the path $\langle \mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3 \rangle$. However, since each elementary transition refers to a specific arc, a transition base cannot impose constraints on arbitrary paths or between paths. This is the job of the transaction base, which is a set of arbitrary transaction formulas.

Definition 4.4 (Models of Transaction Formulas) A path structure, $\mathbf{M}$, is a *path-model* (or simply a *model*) of a $\mathcal{TR}$-formula ϕ, written $\mathbf{M} \models \phi$, if and only if $\mathbf{M}, \pi \models \phi$ for every path π. 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. □

In the models of a transaction formula, dependencies may exist between classical Herbrand structures associated with a path and its subpaths. To see this, consider a simple transaction formula $p \leftarrow q \otimes r$, where p, q and r are ground atomic formulas. Let π be a path in a model of this formula. Split π into two parts, π_1 and π_2. We then have a dependency between three different Herbrand structures on three different paths, π, π_1, and π_2:

$$\text{if} \quad q \in \mathbf{M}(\pi_1) \quad \text{and} \quad r \in \mathbf{M}(\pi_2) \quad \text{then} \quad p \in \mathbf{M}(\pi).$$

That is, if q is true on π_1, and r is true on π_2, then $q \otimes r$ is true on π (by item 5 of Definition 4.2), so p must also be true on π. An informal reading of this situation is that if $q \otimes r$ can execute along path π, then so can p. In an appropriate logic-programming setting, this can be made to mean that p is a *name* of the procedure $q \otimes r$.

4.4 Execution as Entailment

We are now ready to define *executional entailment*, a concept that provides a logical account of transaction execution. Intuitively, execution corresponds to truth on a path. A simplified form of this notion was used in Section 2.

Before presenting formal definitions, we pause to discuss the three distinct parts of $\mathcal{TR}$ logic programs: the transaction base $\mathbf{P}$, the database $\mathbf{D}$, and the 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 procedures for updating the database and answering queries.

The transaction base contains logical rules that define updates and queries. As such, it will normally be composed of formulas containing the serial connectives $\otimes$ or $\oplus$, though classical first-order formulas are also allowed.

In contrast to the transaction base, the database consists entirely of classical first-order formulas. Certain kinds of databases are likely to arise more frequently in practice. The most common databases will be sets of ground atoms (relational databases), function-free Horn clauses (deductive databases), general Horn clauses (logic programs), and disjunctive and existentially quantified formulas (indefinite databases).

Technically, a transition base is nothing but a set of elementary transitions. In practice, however, some transition base will most likely be "hard wired" into the logic programming system at hand, as remarked in Section 3. It follows from our definitions that elementary state transitions are not allowed in transaction bases and databases.

Definition 4.5 (Executional Entailment) Let $\mathcal{B}$ be a transition base, and $\mathbf{P}$ be a transaction base. Let ϕ be a transaction formula, and $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ be a sequence of databases (first-order formulas). Then, the statement

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi \tag{3}$$

is true if and only if $\mathbf{M}, \langle \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \rangle \models \phi$ for every model, $\mathbf{M}$, of $\mathcal{B}$ and $\mathbf{P}$.

Related to this are the statements

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \models \phi \tag{4}$$

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \mathbf{D}_n \models \phi \tag{5}$$

$$\mathcal{B}, \mathbf{P}, \text{---} \mathbf{D}_n \models \phi \tag{6}$$

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 (3) is true. □

Intuitively, Statement (3) 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 (3) is not quite appropriate. To account for this situation, the version of entailment in (4) allows us to omit the intermediate and the final database states. Informally, Statement (4) 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 (4) is not true, we say that transaction ϕ *fails*. In [11, 12], 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 (3) whenever a transaction succeeds.

Statement (6) is the dual of (4). Intuitively, it says that ϕ can execute successfully, *terminating* at state $\mathbf{D}_n$. Statement (5) is also a useful abbreviation that will be used throughout the paper. Informally, 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 4.5.

Lemma 4.6 (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 4.6 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 spelled out in [11, 12], and indeed, they form the model-theoretic basis of these procedures. Item 1 is the basis for a wide class of dynamic constraints, such as those in Section 2.6.

In Lemma 4.6, $n = 0$ 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. Intuitively, this lemma says that the result of evaluating a conjunctive query is the same whether the conjuncts are evaluated sequentially or concurrently.

Lemma 4.7 (Conjunctive Queries) *For any transition base* $\mathcal{B}$*, any transaction base* $\mathbf{P}$*, any database* $\mathbf{D}$*, and any transaction formulas* α *and* β*,*

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

4.5 Examples

To illustrate executional entailment of $\mathcal{TR}$ and the use of Lemma 4.6, we return to Example 2.13, in which a robot simulator moves blocks around a table-top. We assume that the transaction base $\mathbf{P}$ contains the following four rules, adapted from that example:

$$\begin{array}{l}
stackTwoBlocks(X,Y,Z) \ \leftarrow \ move(Y,Z) \otimes move(X,Y) \\
move(X,Y) \ \leftarrow \ pickup(X) \otimes putdown(X,Y) \\
pickup(X) \ \leftarrow \ isclear(X) \otimes on(X,Y) \otimes del{:}on(X,Y) \otimes ins{:}isclear(Y) \\
putdown(X,Y) \ \leftarrow \ X \neq Y \otimes isclear(Y) \otimes ins{:}on(X,Y) \otimes del{:}isclear(Y)
\end{array}$$

For simplicity, our databases are relational, *i.e.*, sets of ground atoms (which is the case for the majority of applications, including most "blocks-world" examples). We also assume that the transition base, $\mathcal{B}$, gives the intended meaning to atoms like *ins:p* and *del:p*. That is, for any ground atom p, the formula $\langle \mathbf{D}, \mathbf{D}' \rangle$ *ins:p* is in $\mathcal{B}$ if and only if $\mathbf{D}' = \mathbf{D} + \{p\}$; and similarly for *del:p*. Thus, the transition base contains infinitely many formulas, including the following three:

$$\Big\langle \{on(a,b), on(b,c), isclear(a)\},\ \{on(b,c), isclear(a)\} \Big\rangle\ del{:}on(a,b)$$

$$\Big\langle \{on(b,c), isclear(a)\},\ \{on(b,c), isclear(a), isclear(b)\} \Big\rangle\ ins{:}isclear(b)$$

$$\Big\langle \{on(b,c), isclear(a), isclear(b)\},\ \{on(b,c), isclear(b)\} \Big\rangle\ del{:}isclear(a)$$

for any blocks a, b, and c.

Finally, all actions in this section start from the following initial database, representing an arrangement of three blocks where $blkA$ is on top of $blkC$, and $blkB$ stands by itself:

$$\mathbf{D}_0 \;=\; \{isclear(blkA), isclear(blkB), on(blkA, blkC)\}$$

Example 4.8 (Picking up a block) Starting from the initial database, $\mathbf{D}_0$, we infer that the robot can pick up $blkA$. During the pickup action, the database passes through an intermediate state, $\mathbf{D}_1$, and ends up at state $\mathbf{D}_2$, where

$$\begin{aligned}\mathbf{D}_1 &= \{isclear(blkA), isclear(blkB)\}\\ \mathbf{D}_2 &= \{isclear(blkA), isclear(blkB), isclear(blkC)\}\end{aligned}$$

Here is the sequence of inferences. The final inference, line 8, states that the action $pickup(blkA)$ successfully takes the database from state $\mathbf{D}_0$ to state $\mathbf{D}_2$ via state $\mathbf{D}_1$.

1. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models isclear(blkA)$, by item 5 of Lemma 4.6.
2. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models on(blkA, blkC)$, by item 5 of Lemma 4.6.
3. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models del{:}on(blkA, blkC)$, by item 4 of Lemma 4.6.
4. $\mathcal{B}, \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \models ins{:}isclear(blkC)$, by item 4 of Lemma 4.6.
5. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models isclear(blkA) \otimes on(blkA, blkC)$, by lines 1 and 2, and item 2 of Lemma 4.6.
6. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models isclear(blkA) \otimes on(blkA, blkC) \otimes del{:}on(blkA, blkC)$,
   by lines 5 and 3, and item 2 of Lemma 4.6.
7. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \models isclear(blkA) \otimes on(blkA, blkC) \otimes del{:}on(blkA, blkC) \otimes ins{:}isclear(blkC)$,
   by lines 6 and 4, and item 2 of Lemma 4.6.
8. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \models pickup(blkA)$, by line 7, the *pickup*-rule, and item 3 of Lemma 4.6. $\square$

4.6 Transaction Answers

So far, we have defined what it means for a formula to be true, which allows us to perform transactions that return yes/no answers. In databases, however, we often want queries and transactions to return a *set* of answers (*e.g.*, all employees who earn more than \$100,000).

For queries, the solution is straightforward: As in classical logic, we can define a *query* to be a first-order formula $\phi(X_0, \ldots, X_k)$ with free variables $X_0, \ldots, X_k$. The answer to this query is the set of variable bindings that makes the formula true. That is, the answer is the following set of tuples:

$$\{\langle c_1, \ldots, c_k\rangle \;\mid\; \mathcal{B}, \mathbf{P}, \mathbf{D} \models \phi(c_1, \ldots, c_k)\}$$

where each c_i is a ground term.

For transactions, the situation is more complex. Because transactions are non-deterministic, the answer can depend on non-deterministic choices made during transaction execution. For instance, continuing Example 2.13, suppose we want our robot to build stacks of blocks, and to return (as an answer) the names of all blocks whose tops are clear *after* the stacks are built. We do not care what blocks are used to build the stacks, so long as these stacks satisfy certain criteria (such as height and color). In this case, the particular blocks affected by the stacking operation is non-deterministic (and is determined by the system at run time), so the answer returned by the transaction is non-deterministic, too.

To make this example concrete, we define a specific stacking operation by adding the following rule to the transaction base:

$$stack(X) \leftarrow move(Y, X) \otimes move(Z, Y)$$

The procedure $stack(X)$ stacks two blocks on top of block X. First, it chooses a block, Y, and puts it on X; then it chooses another block, Z, and puts it on top of Y. The final state of the database is non-deterministic since the choices for the two blocks, Y and Z, are non-deterministic.

Having defined this stacking operation, we ask the robot to stack two blocks on top of $blkA$, and to return the set of blocks that are clear after the operation is complete. This transaction is expressed by the following formula:

$$stack(blkA) \otimes isclear(X) \tag{7}$$

That is, first execute $stack(blkA)$, and then retrieve those values of X that make $isclear(X)$ true. Note that the retrieved values depend on the *final* database state, which is non-deterministic.

The answer to a transaction may depend on the *initial* state of the database as well as the final state. For example, we might ask for those blocks X that are clear *both* before and after the stacking operation. This transaction is expressed by the formula

$$isclear(X) \otimes stack(blkA) \otimes isclear(X)$$

that retrieves the values of X that make $isclear(X)$ true in both the initial and the final states.

More generally, the answer to a transaction may depend on the *intermediate* database states. For example, we might ask the robot to build two stacks, one after the other, and to return those blocks that are clear in between the two stacking operations. This transaction is expressed by the following formula:

$$stack(blkA) \otimes isclear(X) \otimes stack(blkB)$$

This formula commands the robot to stack two blocks on $blkA$, then to retrieve those blocks X that are clear, and then to stack two blocks on $blkB$. Note that here the answers may be different for different execution paths, even if the initial and the final states of these paths are the same.

Thus, unlike database queries, the answer returned by a transaction is not determined by the current database alone. Instead, it may depend on the entire execution path of the transaction. Definition 4.9, below, captures this idea. This simple definition is sufficient for our present purposes. However, as with database queries [17], this definition could be augmented with additional requirements, such as computability and genericity.

Definition 4.9 (Abstract Transactions) An *abstract transaction* is a total mapping from *sequences* of database states to sets of tuples of ground terms. □

Note that database queries are captured by a special case of this definition, the case of sequences of length 1. This is appropriate since queries are a special kind of transaction.

In $\mathcal{TR}$, transactions are specified by a transition base $\mathcal{B}$, a transaction base $\mathbf{P}$, and a transaction formula $\psi(X_1, \ldots, X_k)$ with free variables $X_1, \ldots, X_k$. Specifically, these three elements define an abstract transaction Ψ, where

$$\Psi(\mathbf{D}_0, \ldots, \mathbf{D}_n) \;=\; \{\, \langle c_1, \ldots, c_k \rangle \;\mid\; \mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \;\models\; \psi(c_1, \ldots, c_k) \,\} \tag{8}$$

where each c_i is a ground term, and each $\mathbf{D}_i$ is a database (a first-order formula).

It follows immediately from Definition 4.9 that when applied to an initial database, $\mathbf{D}_0$, the answer returned by a transaction is non-deterministic. Indeed, from $\mathbf{D}_0$, the transaction may execute along any one of many possible paths, each returning a different answer. This idea is captured by the following definition.

Definition 4.10 (Transaction Answers) Let Ψ be an abstract transaction, and let $\mathbf{D}_0, \ldots, \mathbf{D}_n$ be a sequence of databases. Then $\Psi(\mathbf{D}_0, \ldots, \mathbf{D}_n)$ is a possible answer to Ψ at state $\mathbf{D}_0$. $\Box$

Of course, for many sequences of databases, the set (8) will be empty. The transaction does not execute along these paths. It only executes along a path that returns a non-empty answer. If the answer set is empty for every path, then the transaction fails.

4.7 Discussion

Entailment in $\mathcal{TR}$ has certain subtle properties that may not be readily seen from the examples so far. Those examples focus on the special case in which a database is relational and a rule-body is a sequence of atomic actions. (We call this the "serial-Horn" case.) More generally, a database may be an arbitrary first-order formula, and the transaction base may be an arbitrary transaction formula. Furthermore, the transition base may mention only a fraction of all possible databases. In these cases, entailment in $\mathcal{TR}$ can display subtle and important properties. This section discusses some of them.

Classical Conjunction

In Definition 4.5, there may be more than one sequence $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ that starts at $\mathbf{D}_0$. In this case, transactions can succeed in many different ways. This property has two consequences worth mentioning, both concerning classical conjunction.

First, although two transactions ϕ and ψ may succeed individually, their conjunction $\phi \wedge \psi$ may fail, as illustrated in Example 2.16. This is possible because a conjunction requires all its conjuncts to execute along the *same* path.

The second consequence is closely related to the first: It is possible that transactions ψ and $\neg\psi$ may *both* succeed from the same database. Again, the reason is that ψ may succeed along one path, while $\neg\psi$ will succeed along another path. The resolution of this apparent inconsistency is that the transaction $\psi \wedge \neg\psi$ always fails, since it is false on every path.

Example 4.11 (Apparent Inconsistency) Consider the transition base $\mathcal{B}$ of the form

$$\langle \mathbf{D}_1, \mathbf{D}_2 \rangle a \qquad\qquad \langle \mathbf{D}_1, \mathbf{D}_3 \rangle b$$

where $\mathbf{D}_2 \not\equiv \mathbf{D}_3$, and suppose that the transaction base, $\mathbf{P}$, contains the two rules $c \leftarrow a$ and $\neg c \leftarrow b$. Then *both* the following statements are true:

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_1 \text{---} \models\; c \qquad\qquad \mathcal{B}, \mathbf{P}, \mathbf{D}_1 \text{---} \models\; \neg c$$

But the following statement is *false*: $\mathcal{B}, \mathbf{P}, \mathbf{D}_1 \text{---} \models\; c \wedge \neg c$. $\Box$

Inconsistency

The previous example discussed *apparent* inconsistencies in $\mathcal{TR}$. This section discusses *real* inconsistencies. There are three possible sources of inconsistency in $\mathcal{TR}$: the transaction base, the transition base, and the interaction between the two. We shall discuss each one in turn.

The first, and simplest, source of inconsistency is the transaction base $\mathbf{P}$. It may contain an inconsistent combination of formulas, such as $\{b,\ c,\ \neg b \vee \neg c\}$. There is only one path-structure that satisfies such a transaction base—one that assigns the special classical structure, $\top$, to every path. Essentially, it is the user's responsibility to avoid this kind of inconsistency, by writing consistent transaction bases. This is easily accomplished in practice, since modern database and logic programming systems infer only positive facts. They are therefore guaranteed to be consistent. Even the advanced examples developed in [11, 14], which go well beyond current technology, do not infer negative facts, and so do not lead to inconsistency.

The second source of inconsistency is the transition base, $\mathcal{B}$, which determines all possible state transitions in the system. In principle, an elementary transition can mention inconsistent databases (*i.e.*, unsatisfiable first-order formulas), in which case, transactions may enter such states during execution. If s is such a state, the only possible view over it is $\top$, *i.e.*, $\mathbf{M}(s) = \top$ in Definition 4.1. Since $\top$ satisfies all formulas, any transaction entering an inconsistent state (or launched from such a state) immediately succeeds. In this sense, inconsistent states act as "black holes" in the state space of $\mathcal{TR}$. The important point here is that inconsistency is "localized;" that is, inconsistency in just one possible database state *does not* spread to all possible database states, and, thus, inconsistent states *would not* render the entire logic useless.

Even if the transaction base $\mathbf{P}$ and the transition base $\mathcal{B}$ are consistent by themselves, they may not be consistent together. This is the third, and most subtle, source of inconsistency. Suppose, for instance, that the transaction base contains the formula $c \leftarrow b$, and that the current database state contains the atom b. Now suppose that we insert the atom $\neg c$ into the database. The database then contains b and $\neg c$, from which both c and $\neg c$ are derived. In this case, the database (*i.e.*, the stored formula) is consistent, but the view is not. Again, if s is such a state, the only possible view over it is $\top$, and it will thus act as a black hole.

For a vast majority of practical applications, however, the above situations do not arise because modern systems normally deal with positive facts only. For instance, for the situation just described, the transition base would have to allow the insertion of negative literals into the database. That is, it would have to have entries of the form:

$$\langle \mathbf{D}, \mathbf{D} + \{\neg c\} \rangle \ \textit{ins:}\neg c$$

Such updates simply do not occur in modern relational database systems or in classical logic programming systems.[14] Furthermore, as discussed above, logic programming systems do not infer negative facts. Thus, there is no way to insert or to derive a negative fact, so inconsistency is impossible. Even the advanced examples developed in [11, 14] do not lead to inconsistent views, since they do not infer negative facts.

In other applications, however, especially those in AI, the derivation of both positive and negative facts may be important. In such cases, inconsistency is possible, so users must be careful not to build

[14] Of course, one may want to *delete* c, but this is radically different from inserting $\neg c$. The operation of deletion was called *erasure* in [34]. For instance, erasure (deletion) of c from $\mathbf{D}$ yields $\mathbf{D} \vee \mathbf{D}'$, where $\mathbf{D}'$ is the result of inserting $\neg c$ into $\mathbf{D}$. If $\mathbf{D}$ is a set of Horn clauses and c is an atom whose predicate does not occur in the rule heads, then erasure of c yields $\mathbf{D} - \{c\}$.

inconsistent knowledge-bases. This is as true for systems based on first-order logic and modal logic, as it is for systems based on $\mathcal{TR}$. Still, if inconsistency is expected to arise, it would be important to "localize" its effects, so that it will not render the entire knowledge-base useless. Our use of $\top$ addresses this issue. Inconsistency has been addressed in more sophisticated ways in work on para-consistent logics. We mention, in particular, the works [5, 36, 37], since their ideas can be applied to $\mathcal{TR}$ without much difficulty.

5 Hypothetical Reasoning

Hypothetical queries play an important role in reasoning about knowledge. A frequently cited example comes from the British Nationality Act, which states that, "You are eligible for citizenship if your father *would* be eligible *if* he were alive" [24]. The reader is referred to [6, 8, 9] for numerous other examples of hypothetical queries. Because of such queries, it is often necessary to perform hypothetical updates as well as real 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 could perform action α, at which point the action *is* committed. Otherwise, the program could 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.

This section extends the syntax and the semantics of $\mathcal{TR}$ to deal with hypotheticals. A sound-and-complete proof theory is developed in [11, 12], where applications of hypothetical reasoning in $\mathcal{TR}$ are also discussed. One such application is presented at the end of this section.

5.1 Hypothetical Formulas

To express hypotheticals, we introduce a modal operator "$\Diamond$" and a closely related operator "$\Box$". In modal terms, $\Diamond\,\phi$ means that 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. Intuitively, "necessity" means that ϕ is the only transaction that can succeed from the present state. Formally, it 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.

To define the meaning of hypotheticals formally, let $\mathbf{M}$ be a path structure, let s be a state, and let ν be a variable assignment. Then

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

Here, $\langle s\rangle$ is the path of length 1 containing state s. From these definitions, it is easy to verify that the following formulas are tautologies, *i.e.*, that they are true on every path of every structure:

$$
\begin{array}{rcl}
\Diamond\Diamond\phi & \leftrightarrow & \Diamond\phi \\
\Diamond(\phi \vee \psi) & \leftrightarrow & \Diamond\phi \,\vee\, \Diamond\psi \\
\Diamond(\phi \wedge \psi) & \rightarrow & \Diamond\phi \,\wedge\, \Diamond\psi \\
\Diamond(\phi \otimes \psi) & \rightarrow & \Diamond\phi \\
\Diamond(\phi \otimes \psi) & \leftrightarrow & \Diamond(\phi \otimes \Diamond\psi)
\end{array}
\qquad\qquad
\begin{array}{rcl}
\Box\Box\phi & \leftrightarrow & \Box\phi \\
\Box(\phi \wedge \psi) & \leftrightarrow & \Box\phi \,\wedge\, \Box\psi \\
\Box(\phi \vee \psi) & \leftarrow & \Box\phi \,\vee\, \Box\psi \\
\Box(\phi \oplus \psi) & \leftarrow & \Box\phi
\end{array}
\tag{9}
$$

Note that hypotheticals hold immediately, *i.e.*, over paths of length 1 and so they do not cause any real state transitions. This is reflected formally by the following equivalence:

$$\texttt{state} \;\equiv\; \Diamond\texttt{path}$$

where, as before, `path` is the special atom that is true on every path, and `state` is the special atom that is true on every path of length 1 (*i.e.*, at every state). To see that this equivalence holds, note that, by definition, $\Diamond$`path` is true *only* on paths of length 1. Hence, $\mathbf{M}, \pi \models \Diamond\texttt{path}$ if and only if π is path of length 1, *i.e.*, if and only if $\mathbf{M}, \pi \models \texttt{state}$. (It is shown in Section 4.2 that `state` cannot be expressed in $\mathcal{TR}$ without hypothetical operators.) Hypotheticals thus increase the expressive power of our language, since we no longer need the special atom `state`. Henceforth, we shall treat `state` as an abbreviation for $\Diamond$`path`. In fact, since `path` is just an abbreviation for $\phi \vee \neg\phi$, for any ϕ, we no longer need any predefined atoms at all.

Unlike the standard modal operators, our "$\Diamond$" and "$\Box$" are not dual to each other in the normal sense; that is, $\Box\phi \not\equiv \neg\Diamond\neg\phi$ and $\Diamond\phi \not\equiv \neg\Box\neg\phi$. Instead, we have the following two equivalences:

$$\Box\phi \;\equiv\; \texttt{state} \wedge \neg\Diamond\neg\phi \qquad\qquad \Diamond\phi \;\equiv\; \texttt{state} \wedge \neg\Box\neg\phi$$

However, it is not hard to define new hypotheticals for which the usual duality holds [11]. Such operators would be useful for reasoning about program execution, but this is outside the scope of this paper.

5.2 Retrospection

There is another class of modal formulas related to hypotheticals. We call them *retrospective* formulas (abbr., *retrospectives*). Intuitively, a retrospective formula $\phi\Diamond$ tests whether we *could have* executed ϕ to reach the current state. Likewise, $\phi\Box$ tests whether we *must have* executed ϕ to reach the current state.

The semantics of the retrospectives is dual to that of hypotheticals. Formally, let $\mathbf{M}$ be a path structure, let s be a state, and let ν be a variable assignment. Then

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

The following tautologies are analogous to (9):

$$
\begin{array}{rcl}
\phi\Diamond\Diamond & \leftrightarrow & \phi\Diamond \\
(\phi \vee \psi)\Diamond & \leftrightarrow & \phi\Diamond \,\vee\, \psi\Diamond \\
(\phi \wedge \psi)\Diamond & \rightarrow & \phi\Diamond \,\wedge\, \psi\Diamond \\
(\phi \otimes \psi)\Diamond & \rightarrow & \psi\Diamond \\
(\phi \otimes \psi)\Diamond & \leftrightarrow & (\phi\Diamond \otimes \psi)\Diamond
\end{array}
\qquad\qquad
\begin{array}{rcl}
\phi\Box\Box & \leftrightarrow & \phi\Box \\
(\phi \wedge \psi)\Box & \leftrightarrow & \phi\Box \,\wedge\, \psi\Box \\
(\phi \vee \psi)\Box & \leftarrow & \phi\Box \,\vee\, \psi\Box \\
(\phi \oplus \psi)\Box & \leftarrow & \psi\Box
\end{array}
$$

Retrospective formulas are not dual to each other in the usual sense, just as hypothetical operators are not. However, we can define a version of retrospectives that obeys duality, just as we did for hypotheticals.

5.3 An Application: Imperative Programming Constructs

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

$$\begin{array}{l} if\_\Diamond a\_b\_c \longleftarrow (\Diamond a) \otimes b \\ if\_\Diamond a\_b\_c \longleftarrow (\Box\neg a) \otimes c \end{array} \tag{10}$$

The transaction $?-\ if\_\Diamond a\_b\_c$, then, has the effect that if a is possible in the current database state, then b will be executed. Otherwise, if $\neg a$ necessarily holds, then c will be executed. The negation "$\neg$" here is of the negation-by-failure variety. In Section 6, we shall discuss the perfect-model semantics for such negation, which is an adaptation from [54].

In executing transaction $?-\ if\_\Diamond a\_b\_c$ above, the subtransaction $?-\ a$ was executed hypothetically, *i.e.*, without states being changed. If such an execution is possible, then b is executed "for real" and the system may end up in a different state. In this example, the sentence "$\Box\neg a$" is best interpreted as "otherwise;" that is, "otherwise do c" (here, c is done "for real"). The use of "$\Box$" is crucial to the proper formulation of "otherwise do c." In contrast, the "$\Diamond$" in the first clause of (10) could be dropped, leading to another useful form of **if-then-else**:

$$\begin{array}{l} if\_a\_b\_c \longleftarrow a \otimes b \\ if\_a\_b\_c \longleftarrow (\Box\neg a) \otimes c \end{array} \tag{11}$$

The difference between (10) and (11) is that in the latter, the transaction a is done for real (if it is executable), while in the former it is only tested. We call these two conditional statements the "test-version" and the "do-version," respectively.

We shall write **if** $\Diamond a$ **then** b **else** c **fi** for $if\_\Diamond a\_b\_c$ defined by (10), provided that $if\_\Diamond a\_b\_c$ does not occur in the head of any other rule. Likewise, we shall write **if** a **then** b **else** c **fi** for $if\_a\_b\_c$ defined by and (11). Equivalently, these two expressions can be thought of as abbreviations for the formulas $(\Diamond a \otimes b) \vee (\Box\neg a \otimes c)$ and $(a \otimes b) \vee (\Box\neg a \otimes c)$, respectively.

In imperative programming, it is often the case that the **else**-part is omitted, which corresponds to "**else** do nothing." In $\mathcal{TR}$, "do nothing" is tantamount to executing `state`. In this case, the "test-version" of **if** $\Diamond a$ **then** b **fi** can be expressed as follows:

$$\begin{array}{l} if\_\Diamond a\_b \longleftarrow (\Diamond a) \otimes b \\ if\_\Diamond a\_b \longleftarrow (\Box\neg a) \otimes \texttt{state} \end{array} \tag{12}$$

The "do-version" of this operator—one in which a is executed for real—is obtained by dropping "$\Diamond$" from the above.

Similarly, the **while-do** construct has two forms. The "test-version" is

$$\begin{array}{l} while\_\Diamond a\_b \longleftarrow (\Diamond a) \otimes b \otimes while\_\Diamond a\_b \\ while\_\Diamond a\_b \longleftarrow (\Box\neg a) \otimes \texttt{state} \end{array} \tag{13}$$

and the "do-version" is obtained by removing "$\diamond$":

$$\begin{aligned} while\_a\_b &\longleftarrow a \otimes b \otimes while\_a\_b \\ while\_a\_b &\longleftarrow (\Box\neg a) \otimes \texttt{state} \end{aligned} \tag{14}$$

In both cases, $while\_\diamond a\_b$ and $while\_a\_b$ are new propositions. Their definitions are recursive, which is what achieves the iterative effect. Notice, again, the role of "$\Box\neg a$" in the second clause of (13) and (14). 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

$$\textbf{while}\ \ \diamond a\ \ \textbf{do}\ \ b\ \ \textbf{od}$$

for the tail-recursive proposition $while\_\diamond a\_b$ defined by (13), provided that $while\_\diamond a\_b$ does not occur in the head of any other rule. Similarly,

$$\textbf{while}\ \ a\ \ \textbf{do}\ \ b\ \ \textbf{od}$$

is a convenient notation for proposition $while\_a\_b$ defined by (14).

Note that in both **while**-versions, if b fails (*i.e.*, cannot be executed) during some 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 and 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 as follows:

$$\begin{array}{l} \textbf{while}\ \ a\ \ \textbf{do} \\ \quad \textbf{if}\ \ b\ \ \textbf{then}\ \ \texttt{state}\ \ \textbf{else}\ \ \textit{error-handler}\ \ \textbf{fi} \\ \textbf{od} \end{array}$$

Here, if b fails during any iteration, (*i.e.*, if $\Box\neg b$ is true at the current state), then the error-handling transaction is executed. If no special action is required, we would simply substitute `state` (*i.e.*, "do nothing") for "*error-handler*" in the above.

6 Perfect-Model Semantics for Negation as Failure

The theory presented so far is classical in that logical entailment is defined in terms of *all* models of a set of formulas. However, in Logic Programming and in AI, it is common to treat the negation operator as negation-as-failure rather than in the classical sense.

One widely accepted formalization of negation-as-failure is based on the so-called perfect-model semantics [54]. In this section, we adapt this semantics to $\mathcal{TR}$. We use perfect-model semantics because it is two-valued and, therefore, is easy to transplant to $\mathcal{TR}$. In addition, perfect-model semantics lies in the intersection of all major formalizations of negation as failure (for locally stratified programs) and thus is more universally agreed upon.

The first step is to delineate the class of rules we are prepared to deal with.

Definition 6.1 (Generalized-Horn Rules) A *generalized-Horn rule* is a formula of the form $p \leftarrow \phi$, where p is atomic and ϕ is a generalized conjunction. *Generalized conjunctions* are defined recursively as follows:

- If p is an atom, then p and $\neg p$ are generalized conjunctions.
- If ψ is a generalized conjunction, then so are $\Diamond\psi$ and $\psi\Diamond$.
- If $\psi_1, \ldots, \psi_n$ are generalized conjunctions, then so are $\psi_1 \otimes \cdots \otimes \psi_n$ and $\psi_1 \wedge \cdots \wedge \psi_n$.

A *generalized-Horn transaction base* is a finite set of generalized-Horn rules. $\square$

As in classical Horn logic programming, the semantics of negation-as-failure can be problematic when recursion occurs through negation. For instance, since the informal meaning of $\neg p$ is that "p cannot be proved," what is the meaning of the rule $p \leftarrow \neg p$? To avoid such problems, we restrict our attention to locally-stratified transaction bases, which generalize the locally-stratified logic programs defined in [54].

To define local stratification, let $\mathbf{P}$ be a transaction base. $\mathbf{P}^*$ then denotes the ground instantiation of $\mathbf{P}$, *i.e.*, the set of all ground instances of rules in $\mathbf{P}$. Following [54], we construct a directed graph, $\mathcal{D}(\mathbf{P})$, whose nodes are atoms in $\mathbf{P}^*$. The arcs in the graph are defined as follows:

- An *arc* goes from p to q if and only if p occurs in the body of a rule in $\mathbf{P}^*$ that has q in the head.
- An arc that goes from p to q is *negative* if and only if $\neg p$ occurs in the body of a rule in $\mathbf{P}^*$ that has q in the head.

In either case, the literal p or $\neg p$ may appear inside a hypothetical operator. Thus, the following rules each define an arc from p to q that is *not* negative:

$$q \leftarrow p \qquad q \leftarrow \Diamond p \qquad q \leftarrow r \otimes \Diamond(t \wedge \Diamond p)$$

Likewise, the following rules each define a negative arc from p to q:

$$q \leftarrow \neg p \qquad q \leftarrow \Diamond\neg p \qquad q \leftarrow r \otimes \Diamond(t \wedge \Diamond\neg p)$$

Using this graph, we define two partial orders, $\succeq$ and $\succ$. We write $q \succeq p$ if there is a directed path from p to q, and we write $q \succ p$ if there is a *negative* path from p to q. A path is negative if it has at least one negative arc on it. Finally, a transaction base, $\mathbf{P}$, is *locally stratified* if and only if the relation "$\succ$" is well-founded, *i.e.*, it has no infinite sequence of atoms such that $p_1 \succ p_2 \succ \cdots$.

We can now define the perfect models of a transaction base, $\mathbf{P}$, and a transition base, $\mathcal{B}$. They are defined in terms of a *preference* order, $\ll$, on the models of $\mathbf{P}$ and $\mathcal{B}$. Our definition follows the outline of [54]. Let $\mathbf{M}$ and $\mathbf{M}'$ be a pair of Herbrand path models of $\mathbf{P}$ and $\mathcal{B}$. We write $M \ll M'$ if and only if:

For any path, π, and any atom, a, if $M, \pi \models a$ and $M', \pi \not\models a$, then there is a path, π', such that $M, \pi' \not\models b$ and $M', \pi' \models b$, for some atom $b \succ a$. (15)

A *perfect* Herbrand path-model of $\mathbf{P}$ and $\mathcal{B}$, is any Herbrand path-model that is minimal with respect to $\ll$. As in [54], it can be verified that every locally-stratified transaction base has a unique perfect model.

Note that in (15) we compare path-models on two different paths, π and π'. The necessity of this should be clear from the following example. Consider a transaction base, $\mathbf{P}$, that has just one rule, $a \longleftarrow u \otimes \neg b$, and a transition base with just one transition, $\langle \mathbf{D}, \mathbf{D}' \rangle u$. Since perfect models are based on prioritized minimization, intuitively, b should be false wherever possible, after which a's truth is minimized. Therefore, in the perfect model, $\mathbf{M}$, of $\mathbf{P}$ and $\mathcal{B}$, b should be false on all paths; u should be true only on the path $\langle \mathbf{D}, \mathbf{D}' \rangle$; and a should be true only on paths of the form $\mathbf{D}, \mathbf{D}', \ldots$, *i.e.*, on paths whose first arc is $\langle \mathbf{D}, \mathbf{D}' \rangle$.

If, however, the models in (15) were compared on the same paths, we would have had one more minimal model, $\ll$-incomparable with $\mathbf{M}$. In this model, b would be true on all paths starting with $\mathbf{D}'$; u would be true on $\langle \mathbf{D}, \mathbf{D}' \rangle$, as before; and a would be false on all paths. Clearly, this is not the intended model here.

We can now extend the notion of executional entailment to include negation-as-failure. That is, we can define the meaning of statements like

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_1, \ldots, \mathbf{D}_n \models_{\mathsf{perf}} \phi$$

where ϕ is a generalized conjunction and $\mathbf{P}$ is a generalized Horn transaction base. The new definition is identical to Definition 4.5 except that instead of looking at *all* models of $\mathcal{B}$ and $\mathbf{P}$, we look only at their *perfect* models. In the special case where $\mathbf{P}$ and ϕ contain no negation signs, the two definitions are equivalent.

6.1 Perfect-Model Semantics in the Applications

In Section 5.3 we saw one example where negation as failure rather than classical negation was needed in $\mathcal{TR}$. In [11], we also show that temporal constraints on transaction execution, such as "unlock before shooting," also require negation as failure and, thus, a canonic model semantics of the kind described earlier.

However, syntactically, these applications are not covered by Definition 6.1 of generalized-Horn rules. We show, below, that both of these cases can be represented as generalized-Horn rules. For instance, the formula $\Box\neg a$ used to express **if-then-else** and **while-do** is not of the form used in that definition. Similarly, the constraint "unlock before shooting," which, in $\mathcal{TR}$ looks like this:

$$\mathit{unlock} \Leftarrow \mathit{shoot}$$

(which is equivalent to $\mathit{unlock} \oplus \neg \mathit{shoot}$ and $\neg(\neg \mathit{unlock} \otimes \mathit{shoot})$ is not of the right form, since in Definition 6.1 we only allow negation of atoms. However, such a constraint can be represented as a proposition, *unlock-before-shooting*, defined by the following generalized-Horn rules:

$$\begin{array}{rcl} \mathit{unlock\text{-}before\text{-}shooting} & \longleftarrow & \neg \mathit{don't\text{-}unlock\text{-}then\text{-}shoot} \\ \mathit{don't\text{-}unlock\text{-}then\text{-}shoot} & \longleftarrow & \neg(\mathit{unlock} \otimes \mathit{shoot}) \end{array}$$

Note that Definition 6.1 allows atoms, such as *unlock-before-shooting*, to be combined with transactions via classical conjunction, which is how formulas are used as constraints (see Section 2.6).

To express imperative programming constructs, Section 5.3 needed a form of negation that is very similar to the classical idea of negation-as-failure, in which $\neg\psi$ succeeds if and only if ψ fails. As discussed in Section 4.7, however, transactions do not behave this way, even with a perfect-model semantics. The difficulty is that a transaction, ψ, and its negation, $\neg\psi$, can *both* succeed from the same database, since ψ may execute along one path while $\neg\psi$ executes along another path. To overcome this difficulty, Section 5.3 used the necessity operator, $\Box$, since for every database, exactly one of ψ and $\Box\neg\psi$ will succeed. $\Box\neg\psi$ thus corresponds closely to the idea of classical negation-as-failure.

To interpret the formula $\Box\neg\psi$ using our perfect-model semantics, we must reduce it to a generalized-Horn conjunction. This is possible because of a near-dual relationship between necessity and possibility. In particular, as discussed in Section 5,

$$\Box\neg b \ \equiv \ \mathtt{state} \wedge \neg\Diamond b$$

where `state` is the special proposition that is true on paths of length 1, *i.e.*, on states. When b is atomic (which is the important case for imperative programming), the right-hand side of the equivalence can be reduced to a generalized conjunction, $\mathtt{state} \wedge \neg c$, where c is defined by the following generalized-Horn rule: $c \leftarrow \Diamond b$. In this way, imperative programming constructs can be expressed in $\mathcal{TR}$ using our perfect-model semantics.

7 Updating General Deductive Databases

As mentioned previously, the database part of a $\mathcal{TR}$ program may contain Prolog-like rules. It is natural, therefore, to let transactions manipulate database rules in the same way as database facts. This capability has been found useful, for example, in AI programming in Prolog. Unfortunately, rule updates in Prolog are procedural and do not have a logical or declarative semantics. Since $\mathcal{TR}$ allows a database to contain deductive rules, it provides a formal semantics for transactions that perform rule updates. The semantics of each rule insertion and deletion is specified by the transition base, which may reflect any number of semantics that have been studied in the literature (*e.g.*, [29, 58]). This approach works well and is completely general as long as a database is a collection of first-order formulas. However, many logic programs and deductive databases are not purely first-order, since they use negation-as-failure. This section addresses this point.

In Prolog, one writes $assert[a(Y) \ :- \ b(X), c(X,Y)]$ to insert the rule $a(Y) \ :- \ b(X), c(X,Y)$ into the database. Here, the argument of the *assert* command is treated as a first-order *term* that represents an encoding of the corresponding *rule*. This kind of "reflection" of logical formulas into ground terms is well-known in logic, and we shall not dwell on it here. We simply assume that some encoding scheme is given, allowing us to reflect any database rule into a first-order ground term.

Once the encoding mechanism is in place, we can augment our transition base, $\mathcal{B}$, with elementary state transitions of the following form

$$\langle \mathbf{D}, \mathbf{D}' \rangle \ \mathit{ins}{:}[a(Y) \longleftarrow b(X) \wedge c(X,Y)]$$

where the databases $\mathbf{D}$ and $\mathbf{D}'$ are defined appropriately. For databases containing arbitrary first-order formulas, the relationship between $\mathbf{D}$ and $\mathbf{D}'$ can be complex, and has been partly worked out by Grahne, Mendelzon, and Winslett [29, 58]. A simple case is rule insertion into Horn databases. Here, we would add the following formula to the transition base

$$\langle \mathbf{D}, \mathbf{D} \cup \{r\} \rangle \ \mathit{ins}{:}(r')$$

for every Horn database, $\mathbf{D}$, and every Horn rule, r, where r' is the reflection of r.

This approach to updates works for classical first-order databases. In this case, the equivalence relation, "$\stackrel{\mathcal{L}}{\sim}$", of Section 4.1 is normal logical equivalence. That is, we treat databases as indistinguishable if and only if they are equivalent according to classical first-order semantics. However, logic programs that use negation-as-failure are not purely first-order. To allow for such databases, we need a different notion of equivalence. The basic problem is illustrated in the next example.

Example 7.1 (Updating Rule-Bases: a problem) Consider the following two databases:

$$\mathbf{D}_1 \;=\; \{\, a \longleftarrow \neg b \,\} \qquad\qquad \mathbf{D}_2 \;=\; \{\, b \longleftarrow \neg a \,\}$$

From the viewpoint of classical logic, $\mathbf{D}_1$ and $\mathbf{D}_2$ are both equivalent to $a \vee b$. But not so in logic programming, where these two databases mean quite different things. Now, suppose we were to define the elementary transition *ins:*$(\, a \longleftarrow \neg b \,)$. Presumably, it should take us from an empty database to $\mathbf{D}_1$, but *not* to $\mathbf{D}_2$. Thus, the model theory must distinguish $\mathbf{D}_1$ from $\mathbf{D}_2$, even though they are classically equivalent. □

Example 7.1 demonstrates that if we are to allow non-classical databases, as in logic programming, then we have to choose the equivalence relation, $\stackrel{\mathcal{L}}{\sim}$, in Section 4.1 so that it can distinguish databases such as $\mathbf{D}_1$ and $\mathbf{D}_2$ above.

In logic programming and deductive databases, the problem of discriminating between $\mathbf{D}_1$ and $\mathbf{D}_2$ is solved by associating a set of *canonic* models with each database. This set is usually much smaller than the set of all models and often there is only one canonic model per database. One way to define canonic models is to use the perfect-model semantics of [54]. In this section, however, we need not commit ourselves to any specific definition of canonic models. In Example 7.1, the canonic (perfect) model of $\mathbf{D}_1$ makes the proposition a true and b false; whereas the canonic (perfect) model of $\mathbf{D}_2$ does the opposite, making b true and a false.

This suggests the following natural definition: $\phi \stackrel{\mathcal{L}}{\sim} \psi$ if and only if

- ϕ and ψ are classically equivalent; and
- they have the same set of canonic models.

This definition of $\stackrel{\mathcal{L}}{\sim}$ clearly distinguishes between databases that are classically equivalent but have different sets of canonic models. It is also flexible enough to accommodate many different notions of canonic model. Furthermore, it reduces to classical logic if we assume that every model is canonic. (In this sense, classical logic is the weakest theory of canonic models.)

8 Comparison with Other Work

As mentioned earlier, there would appear to be many other candidates for a logic of transactions, since many logics reason about state changes or about the related phenomena of time and action. However, despite the abundance of action logics, none has ever been found suitable for 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. The introduction described a few broad reasons for this unsuitability of existing action logics. This section looks at specific formalisms in more detail. We first discuss declarative languages that were designed for updating databases. We then look closely at action logics presented in the literature of Logic and AI.

As far as databases and logic programming are concerned, we are not aware of any approach to logics for 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 permanently change the current database state. Furthermore, most of the works are limited to updating sets of ground atomic facts.

One feature that sharply distinguishes $\mathcal{TR}$ from all other formalisms is the transition base. The transition base exploits $\mathcal{TR}$'s distinction between stored and derived data. It also provides a completely general approach to elementary updates without sacrificing efficiency. In effect, the transition base serves as an oracle for the proof theory of $\mathcal{TR}$. Thus, many kinds of elementary update can be performed by specialized, efficient algorithms that are commonly used in database systems. Furthermore, many algorithmic features of database systems, such as bulk updates and random sampling, can be relegated to the transition base without sacrificing declarativeness [11, 14]. The same is true of heuristic approximations to intractable problems. Indeed, localizing the *ad hoc* nature of such heuristics to the transition base makes them invisible to the proof theory, the model theory, and the run-time environment of the system. Finally, as explained in [11], elementary transitions need not be recursively enumerable, in general. Testing the emptiness of a predicate in a logic program is one example. In logic programming and deductive databases, such problems are typically dealt with by developing non-monotonic, second-order semantics, and by imposing syntactic restrictions on programs (*e.g.*, [54, 57, 25]). $\mathcal{TR}$'s transition base factors out such problems from the logic, so they can be treated separately. In this way, $\mathcal{TR}$ is able to provide a monotonic, first-order semantics for combining elementary updates (even very nasty ones) into complex transactions.

8.1 Declarative Languages for Database Updates

Winslett [58] did foundational work by supplying the first generally acceptable semantic definition for the result of updating a logical theory. Later on, Grahne, Katsuno and Mendelzon [34, 28, 29] 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.

While [58, 34] take a model-theoretic view of elementary updates, Fagin, Kuper, Ullman, and Vardi [21, 20] approach the problem syntactically. One interesting consequence of this is that, while in [58, 34] an elementary update is always deterministic, in [21, 20] updating a theory with a formula may lead to several alternative states.

Several authors [16, 40, 49] base their approaches to updates on Dynamic Logic [32]. Casanova [16] applies an adaptation of Dynamic Logic to reasoning about concurrent execution of procedural programs that are built out of relational algebra operators plus an explicit relational assignment operator. Updates are done in the hypothetical mode and no complete proof theory is given.

Manchanda and Warren [40] introduce Dynamic Prolog—a logic system where update transactions "work right," *i.e.*, they do not leave a residue in the database when a transaction fails (the residue left by *assert* and *retract* is the most serious problem that plagues Prolog's update mechanism). 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 (which is 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. This is because this proof theory is a verification system: Given an initial state, a final state, and an update procedure, the proof theory can *verify* that the procedure causes the transition; but, given just the initial state and the procedure, it cannot *compute* the final state. In other words, it cannot execute procedures as $\mathcal{TR}$'s proof theory does. Apparently, realizing this drawback, Manchanda and Warren developed an interpreter whose aim was to "execute" transactions. However, this interpreter is incomplete with respect to the model theory and, furthermore, it is not based on the proof theory. To a certain extent, it can be said that Manchanda and Warren have managed to formalize their intuition as a program, but not as an inference system.

Naqvi and Krishnamurthy [48] extended Datalog with update operators, which were later incorporated in the LDL language [49]. Since LDL is geared towards database applications, this extension has bulk updates, for which an operational semantics exists. Unfortunately, the model theory presented in [48, 49] is somewhat limited. First, it matches the execution model of LDL 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 [48, 49] is highly restrictive, making it difficult to build complex transactions out of simpler ones.

Chen has developed a calculus and an equivalent algebra for constructing transactions [18]. Like $\mathcal{TR}$, this calculus uses logical operators to construct actions from elementary updates. There are several differences however. First, unlike $\mathcal{TR}$, Chen's calculus is not a full-blown logic. It extends the domain calculus to include updates, but there is no notion of logical inference. Second, the calculus operates only on relational databases, and not on arbitrary logical theories. Third, the calculus has no analogue of $\mathcal{TR}$'s transition base for specifying elementary updates, so it is restricted to the insertion and deletion of single tuples. However, the most profound difference between the two formalisms shows in the semantics of conjunction. Whereas $\mathcal{TR}$ uses $\wedge$ to express constraints, Chen's calculus uses it to express parallel actions. The main motivation here is that parallel actions make bulk updates easy to express, which is an important database feature. However, there are several disadvantages in the way this is achieved. First, the calculus cannot express the kind of sophisticated constraints that $\mathcal{TR}$ can. At the same time, sequential updates often achieve the same effect as parallel updates. Second, parallel actions greatly complicate the semantics, since they require a minimality principle, which makes the algebra non-monotonic even in the absence of negation. This complication is not called for, since, for instance, $\mathcal{TR}$ achieves the same effect in a much simpler, monotonic way, via so called bulk updates [11, 14]. Third, the syntax is not closed. For instance, negation can be applied to some formulas but not others. In particular, if ψ is an updating transaction, then rules like $p \leftarrow \psi$ are not allowed, since it is equivalent to $p \vee \neg\psi$; indeed, this formula has no meaning in Chen's calculus. It therefore seems unlikely that this calculus can be developed into a full-blown *logic* in a straightforward or satisfying way. Furthermore, the calculus itself is very limited as a programming language, since it has no mechanism for defining recursion or subroutines.

Bry [15] takes a two-tiered approach: first, he uses ordinary queries to determine (generate) the set of atoms to be added/deleted; then, he performs updates meta-logically. Dynamic constraints are also meta-logical; and they involve pairs of different theories. Thus [15] can neither define nor compose complex transactions. It is also impossible to specify constraints on transaction execution or to insert and delete rules. In a sense, [15] provides a *methodology* for expressing what is to be updated rather than a logic for *performing* updates.

Bonner has carried out a thorough analysis of reasoning in Hypothetical Datalog [8, 10, 9, 7, 6], including a sophisticated complexity analysis. Like $\mathcal{TR}$, this work combines elementary updates into complex transactions. Both works also use similar tactics to avoid the infamous frame problem [42]. However, Hypothetical Datalog does not commit updates, does not deal with arbitrary logical databases, and does not combine updates with other logical connectives in arbitrary ways. Instead, Hypothetical Datalog is a disjunction-free logic programming system for relational databases. The semantics of Hypothetical Datalog is also based on different principles than $\mathcal{TR}$'s. In particular, it does not embrace the concept of the execution path, which is fundamental to $\mathcal{TR}$. Consequently, dynamic constraints of the kind discussed in Section 2.6 and in [11] cannot be expressed.

Abiteboul and Vianu developed a family of declarative update languages [1], 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, a facility that any practical programming language must provide. That is, a transaction cannot be given a name, and then be invoked repeatedly from within the language. This lack of subroutines is reflected in the data complexity of some of the languages: they are in PSPACE, whereas recursive subroutines require *alternating* PSPACE, that is, EXPTIME.

The works [51, 19] are related to [1] 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 transactions, constraints, and for reasoning about actions.

8.2 Logics of Action

The work described above is aimed at specifying and *executing* database updates. In contrast, logics of action are aimed at specifying and *reasoning* about actions. Updates are a special case of actions and, in a sense, execution is a special case of reasoning where one infers the final state of the database from the initial state. With this in mind, we examine a number of prominent action logics from the perspective of databases and logic programming.

Generally, these action logics have far greater expressive power than the database languages discussed above. In a sense, however, they are *too* expressive. One lesson of database theory and practice is that if a declarative language (like SQL) is not too expressive, then optimization and high performance are possible. Often, database languages aim for the *minimal* power needed to get a job done. Indeed, most of the languages discussed above have natural implementations that are reasonably efficient and optimizable. In contrast, most of the logics in this section provide no clues about how to efficiently execute transactions. One problem is that they do not distinguish between stored data and derived data. Thus, there is no obvious way to materialize an updated database without also materializing all the logical consequences of the theory. In contrast, state materialization is a central feature of $\mathcal{TR}$. This is, perhaps, most apparent in the proof theory in [11, 12], where each new state is explicitly materialized. There is no analogue of this in other logics.

Although logics of action can have great expressive power, it is sometimes the wrong kind of expressiveness for databases. For instance, since most action logics were not designed as programming languages, important programming features (such as subroutines) are often awkward, if not impossible to express. In addition, action logics often make a sharp distinction between queries and updates. This distinction may be important in some applications, but it is ill-suited for programming. For example, Prolog-style languages and most object-oriented languages treat queries and updates uniformly. A sharp distinction can also be a hindrance to application development. For instance, to debug a query, a programmer may want to insert an update into its definition to monitor its execution or to count the number of times it is called. More generally, as an application evolves, simple queries may develop into more complex transactions with updates, and vice-versa. An abrupt distinction between queries and updates makes this kind of evolution difficult, if not impossible.

$\mathcal{TR}$ makes no such distinction. It treats queries and updates uniformly, as two extreme points on a spectrum of transactions. Like ordinary programs, every formula in $\mathcal{TR}$ can both update the database *and* return an answer. While $\mathcal{TR}$ does not distinguish between queries and updates, it does not force the user to blur the line between the two. If an application calls for such a distinction, it can easily be achieved by using two sorts of predicates, query predicates and update predicates. One way to distinguish the two is to require that all update predicates be prefixed with "*do:*". Thus, *do:enroll(student course)* would be an update, while *grade(student,course)* would be a query. In a similar fashion, deductive databases distinguish two kinds of predicates (base and derived), while the underlying (classical) logic does not.

Finally, most logics of action do not actually perform database updates. Many, in fact, do not even have a notion of database, since they were not intended for database applications. Typically, an action logic describes relationships between different states, without describing how to get from one state to the next. This is fine for reasoning about action; but to actually execute an action, one needs much more information. Unfortunately, large numbers of logical formulas are often needed to completely describe even very simple actions. Even when this is not a problem, however, an action logic is usually just a specification language. Actually executing an action and updating the database must be done outside of the logic. This contrasts with classical logic programming, where executing a query can be carried out by purely logical means, *i.e.*, by the proof theory. Unlike most logics of action, $\mathcal{TR}$ is a language for *both* specifying and executing actions. This is possible because $\mathcal{TR}$ includes logical mechanisms for executing an action and updating the database. This is formalized model-theoretically by the notion of *executional entailment* (Section 4.4). It is even more apparent in the proof theory, in which each elementary update produces a new database state.

Process Logic

As mentioned previously, our path structures are reminiscent of the "path models" in Process Logic [33].[15] However, the meaning and intent of dynamic formulas in [33] is fundamentally different from that of transaction formulas in $\mathcal{TR}$. The former is intended to *reason about* what is true during program execution, while $\mathcal{TR}$ was designed to actually *execute* declaratively specified procedures. This difference in the intent is reflected in the syntax. For instance, Process Logic uses a separate alphabet to represent actions, and a set of modal operators to reason about them. Thus, unlike $\mathcal{TR}$, actions in [33] have a different status than propositions. In particular, actions are not logical formulas, but rather are terms used to construct modal operators. One consequence is that only elementary actions have names. Composite actions cannot be named, and thus the logic lacks a subroutine facility.

[15] A number of different process logics have been proposed in the literature, beginning with Pratt's original work [52]. The version in [33] is closer to $\mathcal{TR}$ than any other incarnation of Process Logic we are aware of.

Another difference between $\mathcal{TR}$ and Process Logic is in the nature of states. As discussed in Section 4, our concept of a state immediately leads to the idea of a transition base—a notion thoroughly missing from all incarnations of the process logic. Finally, although actions can be sequenced in Process Logic, this is not done using a *logical* connective of serial conjunction, as in $\mathcal{TR}$. Consequently, there is no counterpart to $\mathcal{TR}$'s dual connective of serial disjunction, nor to serial implication. Thus, there is no obvious way to express constraints based on these connectives, like the one illustrated in Section 6.1.

Action Logics

Pratt [53] develops a logic, called *Action Logic*, that is similar to, yet different from $\mathcal{TR}$. Like $\mathcal{TR}$, Action Logic does not distinguish between actions and propositions: actions are simply propositions that hold on intervals. The semantics and the intent of the two formalisms are very different however. First, Action Logic is not a language for updating databases or defining transactions. In fact, it has no notion of database, no analogue of $\mathcal{TR}$'s transition base, and no counterpart to executional entailment. Instead, Action Logic is an extension of regular expressions, and as such, it can be viewed as a formalism for defining languages. Second, in contrast to $\mathcal{TR}$'s semantics, which is based on sequences of states, Pratt develops an algebraic semantics, based on *action algebras.* The proof theory for Action Logic is a (finite) set of equations for reasoning about these algebras.

Nevertheless, Action Logic is superficially similar to $\mathcal{TR}$. For instance, it has operators similar to $\otimes$, $\vee$, `state`, and $\neg$`path` in $\mathcal{TR}$; and it has an iteration operator, "*", which can be expressed in $\mathcal{TR}$ using recursion. What makes the comparison seem especially close, is a pair of connectives, denoted $\leftarrow$ and $\rightarrow$ in Pratt's notation, that look very similar to $\mathcal{TR}$'s connectives for serial implication, $\Leftarrow$ and $\Rightarrow$. Semantically, however, these connectives are very different. For instance, the following equation is an axiom of Action Logic:

$$a(a \rightarrow b) \; + \; b \; = \; b \tag{16}$$

Here, + denotes ordinary disjunction, and concatenation denotes sequential composition. This equation is therefore a sequential version of modus ponens: it says that if $a(a \rightarrow b)$ is true, then b is also true. The same is not true in $\mathcal{TR}$. The analogue of equation (16) in $\mathcal{TR}$ is the following statement:

$$a \otimes (a \Rightarrow b) \; \vee \; b \; \equiv \; b$$

However, this statement is *not* a theorem of $\mathcal{TR}$. The intuitive reason is that the two occurrences of a in this statement do not refer to the same time interval. In fact, the second occurrence of a can begin only when the first occurrence ends. Hence, the truth of $a \otimes (a \Rightarrow b)$ does not imply the truth of b.

There is one more difference worth noting. $\mathcal{TR}$ obtains much of its simplicity by having *two* kinds of conjunction, classical and serial. Combined with negation, they lead to two kinds of disjunction, and two kinds of implication, in a natural way. In contrast, Action Logic obtains much of its simplicity by having *one* type of conjunction (serial conjunction), as pointed out in [53]. Action algebras have a lattice-like "meet" operator that might form the model-theoretic basis for another kind of conjunction, but unfortunately, the meet operation is not always defined.

In [56], van Benthem outlines a number of logical approaches to dynamic information processing. These approaches and $\mathcal{TR}$ share the idea that actions are to be represented as propositions, but the similarities end here. The proposed logics are not path-based and there is no notion analogous to the transition base of $\mathcal{TR}$.[16] One of the approaches in [56] is based on a *dynamic interpretation* of

[16] van Benthem actually mentions that a path-based logic would be desirable, but no such logic was developed in [56].

classical predicate logic. A dynamic interpretation associates a pair of states to each proposition, which resembles Dynamic Prolog of Manchanda and Warren [40]. However, the states associated with propositions by the dynamic interpretations of [56] are not database states but rather variable assignments. Van Benthem also discusses an algebraic approach to the logic of dynamic systems, which is akin to the action logic of Pratt [53].

McCarty has outlined a logic of action as part of a larger proposal for reasoning about deontic concepts [43]. His proposal contains three distinct layers, each with its own logic: first-order predicate logic, a logic of action, and a logic of permission and obligation. In some ways, the first two layers are similar to $\mathcal{TR}$, especially since the action layer uses logical operators to construct complex actions from elementary actions. Because of his interest in deontic concepts, McCarty defines two notions of satisfaction. In one notion, called "strict satisfaction," conjunction, $\wedge$, corresponds to parallel action, as it does in Chen's work [18]. In the other notion, called "satisfaction," the same symbol corresponds to constraints, as it does in $\mathcal{TR}$. However, since the focus of this work is on strict satisfaction, the development of path constraints was never considered. Also, there is no analogue of $\mathcal{TR}$'s transition base, and the only elementary updates considered correspond to insertion and deletion of atomic formulas. Although obviously interesting, this action logic was not developed in detail. For instance, although a model theory based on sequences of partial states is presented, there is no sound-and-complete proof theory, and no mechanism is presented for executing actions or updating the database. In contrast, the recent work of McCarty and Van der Meyden [44] is much more detailed, but is very different from $\mathcal{TR}$ and is not intended for updating databases.

The Situation Calculus and Related Works

The situation calculus is a methodology for specifying the effects of actions in first-order predicate calculus. It was introduced by John McCarthy [41] and then further developed by McCarthy and Hayes [42]. To a large extent, the situation calculus is orthogonal to $\mathcal{TR}$: whereas the situation calculus focuses on specifying the effects of primitive actions, $\mathcal{TR}$ focuses on combining such actions into complex ones and executing them. A comprehensive comparison of $\mathcal{TR}$ with situation calculus appears in [11].

The Event Calculus

The event calculus is also a methodology for encoding actions in first-order predicate logic. It was developed by Kowalski and Sergot for reasoning about time and events in a logic-programming setting [39, 38] and for overcoming some of the problems of the situation calculus. Like the situation calculus (and unlike $\mathcal{TR}$), the event calculus makes a sharp distinction between actions and queries. However, inspired by James Allen's treatment of time and action [2], the event calculus focuses on time intervals instead of states. Its use of intervals is somewhat dual to $\mathcal{TR}$'s use of paths: in $\mathcal{TR}$, actions take place over paths, and facts are true at states; but in the event calculus, facts are true over intervals, and actions take place instantaneously. Like the situation calculus, but unlike $\mathcal{TR}$, the event calculus does not store the current database state. Instead, it records a history of events (database transactions), from which it can infer the state of the database at any given time. Furthermore, under certain conditions, it is possible to index the events and achieve efficiency comparable to assert and retract in Prolog [38]. This is possible, for instance, if events are write-only, *i.e.*, if they change the database without reading it, as in "Set Joe's salary to $50,000." When events can both read and write, efficiency is still possible if the database records not only a description of each event, but also its effects. Thus, given the event "Add $5,000 to Joe's salary," the new salary would be computed and stored in the database, in addition to storing the event itself. In this way, salaries would not have to be recomputed from time zero with each salary query.[17] In a logic-programming setting, the event calculus relies on negation-as-failure to solve the frame problem. In contrast, $\mathcal{TR}$ solves the frame problem in a purely monotonic logic, and its proof theory and model theory are both first-order.

Like the situation calculus, all actions in the event calculus are elementary. Thus, there is no way to define complex actions in terms of simpler ones. A user can specify that one event occurs before another, but he cannot construct transactions using conditional statements, iterative loops, and subroutines. From the database perspective, however, it seems that the event calculus was not intended as a language for building complex transactions. Rather, it can be seen as a logical account of database logs. By centering on semantically meaningful events, it provides flexible support for historical databases.

Thus, the event calculus seems to be complementary to $\mathcal{TR}$. Indeed, since $\mathcal{TR}$ includes all of first-order logic as a special case, this calculus can be incorporated in $\mathcal{TR}$ as a methodology for reasoning about events and supporting historical databases. Most database applications, however, are not historical, and in these cases, $\mathcal{TR}$ supports updates without the added cost that historical databases demand. $\mathcal{TR}$ can also be of benefit to the event calculus, as it provides a logical mechanism for inserting events and their effects into the database, for revising incorrect assertions, and for combining simple events into complex transactions—a mechanism that is ostensibly lacking in [38].

Temporal Logics for Reasoning about Action

In addition to the situation and event calculi, there are other approaches to formulating action within first-order logic (*e.g.*, [2, 45, 31]). Generally, these approaches are concerned with difficult problems in Artificial Intelligence and Linguistics. Often they are concerned with the the interaction of time and action with belief, intention and causality. As such, the number and the complexity of the axioms for these systems is quite large compared to the proof theory of $\mathcal{TR}$. More importantly, developing a declarative programming language (like $\mathcal{TR}$) for specifying and executing transactions is not their main concern. See [11] for a more thorough comparison with these works.

In [47], Moszkowski develops an imperative programming language called *Tempura* based on Interval Temporal Logic (ITL) [30, 46]. Moszkowski shows how each Tempura statement translates into ITL, shows how to specify numerous applications in Tempura, and describes an interpreter for executing Tempura programs. There are important similarities between Tempura and $\mathcal{TR}$. For instance, Tempura emphasizes the *execution* of programs specified in logic. Tempura also evaluates formulas on paths, that is, on sequences of states; and it has an operator of sequential composition, called *chop*, which is identical to serial conjunction in $\mathcal{TR}$. Finally, like serial-Horn $\mathcal{TR}$, Tempura represents a subset of a larger logic. However, there are also many significant differences between Tempura and $\mathcal{TR}$, which we itemize below.

- Tempura is not a *database* programming language. Indeed, there is no notion of database or of persistence in Tempura or in ITL, and there is no analogue to $\mathcal{TR}$'s transition base. Furthermore, if applied to databases, Tempura and ITL would both suffer from the frame problem and the ramification problem [42, 23].

[17] R. Kowalski, personal communication

- Although it has a logical semantics, Tempura is an *imperative* programming language, not a *logic* programming language. Consequently, Tempura has no built-in facilities for unification, for backtracking, or for non-determinism.

- Although a notion of serial conjunction is defined, the dual notion of serial disjunction is not considered, nor is serial implication. Consequently, the kind of dynamic constraints described in Section 2.6 and in [11] were not developed.

- Tempura represents an executable fragment of ITL, but a fragment without a clean, simple characterization. In particular, Tempura is not a "Horn" fragment of ITL.

- Unlike serial-Horn $\mathcal{TR}$ (and classical Horn logic), Tempura lacks an efficient SLD-style inference system. In fact, proof theory is barely mentioned in [47].

- Although Tempura has subroutines, they cannot be expressed within ITL [47]. It may be possible to extend ITL to include (recursive) subroutines, perhaps by adding a fixpoint operator, but [47] provides no details.

Finally, it is worth comparing $\mathcal{TR}$, ITL and Process Logic (discussed earlier). As in $\mathcal{TR}$ and Process Logic, formulas in ITL are true on paths, that is, on sequences of states. However, the semantics of ITL and Process Logic are fundamentally different from that of $\mathcal{TR}$. For instance, in both ITL and Process Logic, an atom is true on a path iff it is true at the first state of the path. In contrast, in $\mathcal{TR}$, the truth of atoms on a path is interpreted as *execution* and is independent of the states that make up the path. Such dependencies can be described in $\mathcal{TR}$ by logical axioms, but they are not built into the semantics. One consequence is that in ITL and Process Logic, an atom can only represent a static fact, such as $age(gorby, 60)$, whereas in $\mathcal{TR}$, an atom can represent an action, such as $promote(gorby, president)$. This is one reason why ITL must be extended in order to define subroutines, since there is no facility for assigning names to complex actions.

Procedural Logic

Georgeff and Lansky develop a formalism called Procedural Logic whose intent is very similar to that of $\mathcal{TR}$ [27, 26]. Both systems focus on executable specifications of transactions, both provide a subroutine facility, both allow for arbitrary primitive actions, both allow non-deterministic actions, and both provide some kind of constraints. Indeed, we are in complete agreement with Georgeff and Lansky on the limitations of existing logics of action. However, there are important differences between Procedural Logic and $\mathcal{TR}$, both practically and philosophically. These difference arise largely because Procedural Logic is actually two separate things: An implementation (called PRS), which provides the aforementioned functionality; and a formalization, which accounts for only a fraction of this functionality. It is probably fair to say that $\mathcal{TR}$ is a formalization that accounts for almost all of PRS and more. Further comparison can be found in [11].

9 Conclusions

We have reviewed the main ideas underlying Transaction Logic, a novel declarative formalism for specifying and executing actions. A sound-and-complete proof theory and a wide range of applications were alluded to, but their full treatment appears elsewhere [11]. 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 many others.

Acknowledgments: Alberto Mendelzon provided us with many insights regarding updates of logic theories. Thanks to Ray Reiter for commenting on various aspects of this paper, especially on the issues related to the frame problem. Mariano Consens took a very close look at the first draft and many improvements are due to his comments. Comments by Greg Meredith on Pratt's Action Logic, and discussions with Gösta Grahne, Peter Revesz, Fangzhen Lin, Javier Pinto, Dimitris Lagouvardos, Jan Van den Bussche, Jan Chomicki, Roel Wieringa, and Ron van der Meyden are also gratefully acknowledged.

References

[1] 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.

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

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

[4] 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.

[5] H.A. Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[6] A.J. Bonner. A logic for hypothetical reasoning. In *National Conference on Artificial Intelligence (AAAI)*, St. Paul, Minn., August 1988. AAAI Press, Menlo Park, CA.

[7] A.J. Bonner. Hypothetical Datalog: Negation and linear recursion. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 286–300, Philadelphia, PA, March 29–31 1989. ACM.

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

[9] A.J. Bonner. *Hypothetical Reasoning in Deductive Databases.* PhD thesis, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, USA, October 1991. Published as Rutgers Technical Report DCS-TR-283.

[10] A.J. Bonner. Hypothetical Reasoning with Intuitionistic Logic. In R. Demolombe and T. Imielinski, editors, *Non-Standard Queries and Answers*, Studies on Logic and Computation, chapter 8, pages 187–219. Oxford University Press, October 1994.

[11] 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*.

[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] A.J. Bonner and M. Kifer. A general logic of state change. Technical report, CSRI, University of Toronto, 1995. In preparation.

[14] A.J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In A. Ohori C. Beeri and D.E. Shasha, editors, *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, Workshops in Computing, pages 309–337. Springer-Verlag, February 1994. Workshop held on Aug 30–Sept 1, 1993, New York City, NY.

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

[16] M.A. Casanova. *The Concurrency Control Problem for Database Systems*, volume 116 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1981.

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

[18] 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.

[19] 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.

[20] R. Fagin, G.M. Kuper, J.D. Ullman, and M.Y. Vardi. Updating logical databases. In P.C. Kanellakis, editor, *Advances in Computing Research*, volume 3, pages 1–18. Plenum Press, 1986.

[21] R. Fagin, J.D. Ullman, and M.Y. Vardi. On the semantics of updates in databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 352–365, New York, 1983. ACM.

[22] 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.

[23] J. Finger. *Exploiting Constraints in Design Synthesis*. PhD thesis, Stanford University, 1986.

[24] D.M. Gabbay and U. Reyle. N-prolog: An extension of Prolog with hypothetical implications. I. *Journal of Logic Programming*, 1(4):319–355, 1984.

[25] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.

[26] M.P. Georgeff and A.L. Lansky. Procedural knowledge. In *Proc. IEEE Special Issue on Knowledge Representation*, volume 74, pages 1384–1398, 1986.

[27] M.P. Georgeff, A.L. Lansky, and P. Bessiere. A procedural logic. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 516–523, San Francisco, CA, August 1985. Morgan Kaufmann.

[28] G. Grahne. Updates and counterfactuals. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 269–276, Boston, Mass., April 1991.

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

[30] J.Y. Halpern, Z. Manna, and B.C. Moszkowski. A high-level semantics based on interval logic. In *International Conference on Automata, Languages and Programming (ICALP)*, pages 278–291. Springer-Verlag, 1983. Number 154 in *Lecture Notes in Computer Science*.

[31] J.Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. In *Intl. Symposium on Logic in Computer Science (LICS)*, pages 279–292, 1986.

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

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

[34] 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.

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

[36] M. Kifer and E.L. Lozinskii. RI: A logic for reasoning with inconsistency. In *4-th Intl Symp. on Logic in Computer Science*, pages 253–262, June 1989.

[37] M. Kifer and E.L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, November 1992.

[38] R.A. Kowalski. Database updates in event calculus. *Journal of Logic Programming*, 12(1&2):121–146, January 1992.

[39] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[40] 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.

[41] J. McCarthy. Situations, actions, and clausal laws, memo 2. Stanford Artificial Intelligence Project, 1963.

[42] 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.

[43] L.T. McCarty. Permissions and obligations. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 287–294, San Francisco, CA, 1983. Morgan Kaufmann.

[44] L.T. McCarty and R. van der Meyden. Reasoning about indefinite actions. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 59–70, Cambridge, MA, October 1992.

[45] D. McDermott. A temporal logic for reasoning about plans and processes. *Cognitive Science*, 6:101–155, 1982. Reprinted in *Readings in Planning*, J. Allen, J. Hendler, A. Tate (eds.), Morgan-Kaufmann, 1990, 436–463.

[46] B.C. Moszkowski. *Reasoning about Digital Circuits*. PhD thesis, Department of Computer Science, Stanford University, July 1983. Available as Technical Report STAN-CS-83-970.

[47] B.C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, U.K., 1986.

[48] 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.

[49] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, Rockville, MD, 1989.

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

[51] 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.

[52] V.R. Pratt. Process logic. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 93–100, January 1979.

[53] V.R. Pratt. Action logic and pure induction. In *Workshop on Logics in AI*, volume 478 of *Lecture Notes in Computer Science*, pages 97–120. Springer-Verlag, 1990.

[54] 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.

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

[56] J. van Benthem. *Language in Action: Categories, Lambdas and Dynamic Logic*. Elsevier Science Pub. Co., Amsterdam, New York, 1991.

[57] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 221–230, New York, 1988. ACM.

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