A Transaction Logic for Database Specification

Xiaolei Qian
Department of Computer Science, Stanford University
and Kestrel Institute

Richard Waldinger
Department of Computer Science, Stanford University
and Artificial Intelligence Center, SRI International

Abstract

We introduce a logical formalism for the specification of the dynamic behavior of databases. The evolution of databases is characterized by both the dynamic integrity constraints which describe the properties of state transitions and the transactions whose executions lead to state transitions. Our formalism is based on a variant of first-order situational logic in which the states of computations are explicit objects. Integrity constraints and transactions are uniformly specifiable as expressions in our language. We also point out the application of the formalism to the verification and synthesis of transactions.

1. Introduction

A database contains a collection of data that models some part of the world. The data in a database is grouped into several relations. A database schema describes the semantics of the data stored in the database by specifying the structure of the relations and the relationships among the data in different relations at different times of the world evolution. We can view a database schema as a first-order theory which characterizes both the static properties of database states and the dynamic properties of state transitions. The collection of valid database states and state transitions is therefore an interpretation of the theory. Application-specific axioms of the theory are called integrity constraints. Integrity constraints represent the time-invariant properties of data in the database and serve as the validity criteria of the database evolution. In order to be a model of the evolving world, a database system must handle changes and check, when a state transition occurs, that both the new state and the state transition are valid with respect to the integrity constraints.

A transaction is a program of actions which manipulates data in the database. In general a state transition is caused by executing a transaction on a valid database state to yield another state. The transactions we consider are deterministic programs such that the resulting state of performing a transaction in a state is uniquely determined by the initial state and the transaction. We also want to have transactions correspond to real-world actions. While physically any update sequence is executable in the database, not all of them have meaningful counterparts in the real-world for which the database is a model. The evolution of a database can be depicted by a directed graph whose nodes are database states and whose arcs are transactions. This evolution graph has the following properties: (1) it is not complete, i.e., not every state is reachable from every other state via the execution of a transaction; (2) it is a multi-graph, i.e., there may be more than one transaction capable of transforming one state to another; and (3) it is both reflexive and transitive, because every state is reachable from itself through a null transaction and the concatenation of two transactions is also a transaction. Compared with general Kripke structures, database evolution graphs characterize not only possible worlds but also possible actions and therefore bear more semantic information.

There are various ways that the evolution graph can be constrained with the integrity constraints. Static constraints specify the semantics of single states in the graph. Dynamic constraints specify the semantics of collections of states and transactions. We are especially interested in a subclass of the dynamic constraints called the *transaction constraints*, which describe the relationships among two states and a transaction that connects them. The verification of integrity constraints involves the consistency proof of the set of constraints, i.e., a proof of the existence of a non-empty evolution graph that satisfies the constraints. The validation of a transaction against a set of integrity constraints involves the proof that the transaction preserves the validity of the integrity constraints. We need a formal system in which we can conduct such verification and validation conveniently, efficiently, and automatically. In such a formal system, computational states and state transitions should be an integral part of the vocabulary, integrity constraints and transactions should be specifiable as expressions, transactions should correspond to programs executable in the databases, and the effects of transactions on the validity of the integrity constraints should be derivable from formal proofs.

There has been extensive work in the specification and validation of integrity constraints, most of which

concentrated on static constraints with first-order logic as the underlying formalism. The importance of the specification of the dynamic database behavior is realized in [2,3]. Research in dynamic constraints is dominated by the use of temporal logic[6,8,11] or higher-order logics[7]. There are three problems associated with temporal logic approach. First, temporal logic is natural in describing executions of transactions, rather than transactions themselves. New variables have to be introduced in order to refer to objects at different states implicitly. Secondly, transactions are not expressible in temporal logic. Although several researchers noticed the importance of expressing transactions[6,11], they treat the set of transactions as a given set of "black-box" actions whose effects on the validity of constraints are pre-determined and cannot be reasoned about. The third problem of using temporal logic is associated with the complexity of its proof methods. It is already known that first-order temporal logic has no effective complete proof methods[1]. Automated theorem-proving techniques for temporal logic are less developed than those for classical first-order logic, giving fewer opportunities for automated constraint validation. The expressive power of temporal logic is severely restricted in [8] to make the validation of dynamic constraints possible.

There is growing interest in the automatic verification of transaction validity with respect to a set of integrity constraints. Variations of dynamic logic[5,9,12] or Boyer-Moore logic[19] are used as verification formalisms. Transactions are objects in the logic and the effects of transactions are determined by axioms, built-in lemmas, and inference rules which describe the effects of various language constructs. In order to reason about expressions in the transaction language, first-order logic is usually combined with dynamic logic in incoherent ways. Theorems involving existential quantification cannot be handled in Boyer-Moore logic. Dynamic constraints are not specifiable in these formalisms directly because states are not objects in the language. Hence only the validity of transactions against static constraints can be verified. These logics are not suitable for transaction synthesis.

Situational logic was introduced by McCarthy[18] as a convenient formalism for describing situations, actions, and causality. Burstall[4] first used this formalism to reason about programs that manipulate the states of a computation. It has been used in QA3[10] to synthesize robot plans and in PROW[21] to synthesize imperative programs. Manna and Waldinger adapted situational logic to be a general framework for describing ALGOL-like programming languages[14]. Recently a restricted variant of situational logic is developed for automated planning[15] and for the deductive synthesis of imperative LISP programs[16]. This variant has been shown to be a very useful and feasible formal system for the automated program synthesis.

We introduce a transaction logic adapted from a variant of first-order situational logic and apply it to the specification of database integrity constraints and transactions. In our logic, computational states and state transitions are explicit objects. Integrity constraints are logical expressions which may refer to states and state transitions. Transactions are expressions mapping from states into states. By constraining ourselves to classical first-order logic which has a more efficient proof theory and better developed automated theorem-proving techniques than higher-order logics, the verification of database specification is reduced to first-order consistency proofs. Showing that a transaction preserves a set of integrity constraints is equivalent to testing the satisfaction of a sentence, and the synthesis of a transaction involves a constructive proof of a theorem in our logic.

The paper is organized as follows. The transaction logic is introduced in Section 2. Relational databases, schemas, transactions, and integrity constraints are defined in terms of the logic in Section 3. In Section 4 we apply the formalism to the specification of integrity constraints and transactions respectively. Section 5 concludes the paper.

2. A Transaction Logic

We motivate the definition of our transaction logic by considering its expected properties. First of all, states and state transitions should be an integral part of the vocabulary, which means that they have to be explicit objects in our language. States should characterize the space of the database evolution. State transitions are viewed as mappings from states into states, which can be composed to form new state transitions. Secondly, integrity constraints and transactions should be specifiable as expressions. This requires that the expressions in the logic can refer to different states and state transitions. In order to reason about the effects of transactions on the validity of the integrity constraints, an appropriate proof system for our transaction logic is necessary. Finally, transactions should only correspond to executable state transitions. As an example of non-executable transactions, consider the following program as a sentence in ordinary situational logic:

$$\begin{aligned}&\textbf{if } \mathit{greater\text{-}than}(\mathit{modify}(s_0, \mathit{sal}(c), \mathit{sal}(c) + 100),\\ &\qquad\qquad \mathit{sal}(c), \mathit{sal}(\mathit{mgr}(c)))\\ &\textbf{then } \mathit{modify}(s_0, \mathit{sal}(c), 1.1 * \mathit{sal}(c))\\ &\textbf{else } \mathit{modify}(s_0, \mathit{sal}(c), 1.2 * \mathit{sal}(c))\end{aligned}$$

The above transaction first increases the salary of c at state s_0 by 100 to yield a new state, then tests whether

the salary of c is greater than the salary of the manager of c in the new state; if it is true then the salary of c is increased by 10% of its original salary at s_0, otherwise it is increased by 20%. The program is not executable because as soon as the salary of c is increased by 100, the original salary value is destroyed and there is no way to execute either of the branches. The problem lies in the fact that computer memory represents implicitly the current state of the computation and therefore programs only have access to this current state. To avoid non-executable transactions, only a subset of the expressions of the language should be considered as sound transactions, although the full power of the situational logic should be retained as specification and proof mechanisms.

We define our transaction logic to be an instance of many-sorted first-order logic. We distinguish two classes of sorts in our logic: the *situational sorts* and the *fluent sorts*. Each situational sort has an associated fluent sort and vice versa. There are 5 types of fluent sorts and the associated situational sorts: (1) the *state* sort **state**; (2) the *atom* sort of natural numbers **at**, which is the sort for attribute values; (3) the *n-ary tuple* sort **ntup** for every $n > 0$, which are the sorts for tuples in relations; (4) the finite *n-ary set* sort **nset** for every $n > 0$, which are the sorts for relations; and (5) the *identifier* sorts, which are the sorts for tuple and relation identifiers, including the *n-ary tuple identifier* sort **nt-id** and the *n-ary set identifier* sort **ns-id**.

We also distinguish two kinds of expressions: (1) expressions of situational sorts are *situational expressions* or *s-expressions*, which denote particular values in specific states; and (2) expressions of fluent sorts are *fluent expressions* or *f-expressions*, which only have values when they are evaluated at certain states. In other words, they can be viewed as mappings from states to values, which *assign meanings* to states. As a notational convention, we'll write e for an f-expression and e' for its associated s-expression. For example, the s-expressions

$$\mathit{salary}'(w, e'),\ \mathit{work\text{-}in\text{-}project}'(w, e', p'),\ \mathit{hire}'(w, e'),$$

where w, e', p' are also s-expressions, denote respectively the salary of employee e' at state w, the truth value of the assertion "employee e' works in project p' at state w", and the state obtained after hiring person e' at state w. On the other hand, the f-expressions

$$\mathit{salary}(e),\ \mathit{work\text{-}in\text{-}project}(e, p),\ \mathit{hire}(e),$$

where e, p are also f-expressions, only denote an object (salary), a truth value (e works in p), and a state (after hiring e) when they are evaluated in a particular state.

Since f-expressions are mappings from states into objects, truth values, or states, we can compose them to form new f-expressions. We consider three such *fluent functions*. For a variable x, an f-formula p, and f-expressions s, t of state sort, we define the f-expressions $s;;t$, **if** p **then** s **else** t, **foreach** $x|p$ **do** s to be respectively the composition fluent, the condition fluent, and the iteration fluent. The composition fluent $s;;t$ represents the fluent obtained by evaluating s followed by t. The condition fluent **if** p **then** s **else** t denotes the fluent obtained by evaluating s if p is true and evaluating t otherwise. Let $x_1, \ldots, x_n$ be an arbitrary enumeration of all the x's that satisfy p, the iteration fluent **foreach** $x|p$ **do** s is equivalent to the composition of n fluents: $s[x_1/x];;\ldots;;s[x_n/x]$. The result of evaluating an iteration fluent at a state w is not defined if either there are infinite number of x's satisfying p at w or the resulting state depends on the order of enumeration. The composition fluent is associative, and has an identity Λ which is an f-constant denoting the identity mapping:

$$(s;;t);;u = s;;(t;;u) \qquad \textit{composition-associativity}$$
$$\Lambda;;s = s;;\Lambda = s \qquad \textit{identity-fluent}$$

The f-expressions do not refer to states explicitly. In order to determine the object, truth value, or state that an f-expression e designates with respect to a specific state w, we provide *situational functions* and apply them to w, e. The s-expressions $w{:}e$, $w{::}e$, $w;e$ denote respectively the object, the truth value, and the state obtained by evaluating e at state w. For example, the s-expressions

$$w{:}\mathit{salary}(e),\ w{::}\mathit{work\text{-}in\text{-}project}(e, p),\ w;\mathit{hire}(e),$$

denote respectively the salary of employee e at state w, the truth value of the assertion "employee e works in project p" at state w, and the state after hiring e at state w. The following axioms govern the behavior of the situational functions. For a state w, an f-function $f(t_1, \ldots, t_n)$ of an object sort, an f-function $g(t_1, \ldots, t_n)$ of state sort, an f-predicate $P(t_1, \ldots, t_n)$, and an f-formula $p(\bar{x}, \bar{y})$, where $t_1, \ldots, t_n$ are f-terms, $\bar{x} = [x_1, \ldots, x_m]$ and $\bar{y} = [y_1, \ldots, y_n]$ are two lists of the free variables of p,

$$w{:}f(t_1, \ldots, t_n) = f'(w, w{:}t_1, \ldots, w{:}t_n) \qquad \textit{object-linkage}$$
$$w{::}P(t_1, \ldots, t_n) = P'(w, w{:}t_1, \ldots, w{:}t_n) \qquad \textit{predicate-linkage}$$
$$w;g(t_1, \ldots, t_n) = g'(w, w{:}t_1, \ldots, w{:}t_n) \qquad \textit{state-linkage}$$
$$w{:}\{f(\bar{y})|p(\bar{x}, \bar{y})\} = \{f'(w, \bar{y})|p'(w, \bar{x}, \bar{y})\} \qquad \textit{setformer-linkage}$$

The interaction of the situational functions with the fluent functions are also characterized by a set of axioms, which relate the evaluation of composite fluents with the evaluation of the component fluents. For state w, f-terms s, t of state sort, and f-formula p,

$$w;(s;;t)=(w;s);t \qquad \textit{composition-linkage}$$

$w;(\textbf{if } p \textbf{ then } s \textbf{ else } t) = (\mathit{if}\ w{::}p\ \mathit{then}\ w;s\ \mathit{else}\ w;t)$ *condition-linkage*

$w;(\textbf{foreach } x|p \textbf{ do } s) = w;s_n$ *iteration-linkage*

where $\{x|w{::}p\} = \{x_1, \ldots, x_n\}$ is the set of x's satisfying $w{::}p$, and $s_n = s[x_1/x];;\ldots;;s[x_n/x]$ is the composition of n f-terms.

Besides situational functions and fluent functions, there are four kinds of functions and predicates in our logic: (1) functions and predicates over natural numbers, including $+, max, min, sum, size_n, \leq$; (2) functions over n-ary tuples, including *selector* $select_n$, *generator* $tuple_n$; (3) functions and predicates over sets of n-ary tuples, including *union* $\cup_n$, *intersection* $\cap_n$, *set difference* $\sim_n$, *cartesian product* ${}_m\times_n$, *set formers* $\{f(\bar{y})|p(\bar{x},\bar{y})\}$ where $\bar{x} = [x_1, \ldots, x_m]$, $\bar{y} = [y_1, \ldots, y_n]$ are two disjoint lists of the free variables of p and $\bar{y}$ is the list of free variables of f, *membership* $\in_n$, *subset* $\subseteq_n$; (4) state-changing functions, including $insert_n$ for inserting an n-ary tuple into a set (relation) of n-ary tuples, $delete_n$ for deleting an n-ary tuple from a set (relation) of n-ary tuples, $modify_n$ for modifying an attribute of an n-ary tuple, *assign* for creating a new relation; and (5) an *identifier function id* which gives the identifier of a tuple or relation. New functions can be (recursively) defined in terms of these built-in functions. For every f-function symbol f, we have an associated s-function symbol f' which takes an extra state argument. For example, $hire(e)$ is an f-function, while $hire'(w, e')$ is the associated s-function. Predicates are handled in similar ways.

Since only s-expressions denote values, we define axioms in our transaction logic to be s-formulas. We utilize an appropriate set of axioms for natural numbers, n-ary tuples, and finite sets to describe the functions and predicates associated with these data structures, such as the axioms of Presburger Arithmetic and the axioms of sets[17]. In order to reason about the effect of evaluating state-changing fluents, we need two groups of axioms. The *action axioms* specify the parts of the states which are changed as the result of evaluating the fluents and how they are changed. The *frame axioms* specify the exact scope of the changes, i.e., what other parts of the states that do not change. For example, the fluent $modify_n(t, i, v)$ can be described by the following action axiom and frame axiom:

$(1 \leq i \leq n) \rightarrow (select'_n(modify'_n(w, w{:}t, i, v),$
$\quad modify'_n(w, w{:}t, i, v){:}t, i) = v)$ *modify-action*

$(i \neq j \vee id'(w, w{:}t_1) \neq id'(w, w{:}t_2)) \rightarrow$
$\quad (select'_n(w, w{:}t_1, i) = select'_n(modify'_n(w, w{:}t_2, j, v),$
$\quad\quad modify'_n(w, w{:}t_2, j, v){:}t_1, i))$ *modify-frame*

The set of these axioms about the data structures, state-changing fluents, situational functions, and fluent functions form a domain-independent theory of the database evolution in classical many-sorted first-order logic. We call this theory the *situational transaction theory* and denote it by $\mathcal{T}_\mathcal{L}$ where $\mathcal{L}$ is the underlying logical language. The theory can be extended with domain-dependent axioms or integrity constraints to form theories for specific applications. A first-order proof system such as the deductive tableau system in [13] is sufficient for performing deduction in this theory.

3. Relational Databases

The situational transaction theory introduced in Section 2 characterizes the domain-independent properties of databases, both the evolution of databases and the structures of data at every database state. The domain-dependent properties of a relational database, those representing the semantics of the data, are characterized by a set of relation schemas together with a set of integrity constraints. If we define a relational database schema intuitively to be a collection of knowledge about the database, it should contain both the domain-independent and domain-dependent properties of the data. This leads to the following definition.

Definition 1. A *relational database schema* Σ is a triple $\langle \mathcal{T}_\mathcal{L}, \mathcal{R}, IC \rangle$, where $\mathcal{L}$ is the situational language, $\mathcal{T}_\mathcal{L}$ is the situational transaction theory, $\mathcal{R}$ is a set of relation f-constants, and IC is a set of closed s-formulas over $\mathcal{L} \cup \mathcal{R}$, called the *integrity constraints*.

Any relational database can be viewed as a model of the situational transaction theory. The model captures the whole lifetime of the database. In such a model, there is a set of computational states. Each state is characterized by the values of objects in the database: attributes, tuples, relations, etc. Fluent expressions correspond to mappings from states to objects. Hence we have:

Definition 2. A *relational database* DB_Σ over the relational database schema $\Sigma = \langle \mathcal{T}_\mathcal{L}, \mathcal{R}, IC \rangle$ is a model of the situational transaction theory $\mathcal{T}_\mathcal{L}$ over the language $\mathcal{L} \cup \mathcal{R}$. A *database state* is an element in the universe assigned to the situational state sort in $\mathcal{L}$.

As mentioned before, only a subset of the expressions in our transaction logic corresponds to executable programs. This subset is exactly the set of fluent terms. Therefore programs do not explicitly designate objects but they yield objects when being evaluated or executed in specific states. There are two types of f-terms or programs: f-terms of object sorts and f-terms of state sort. The definition below makes these concepts clear.

Definition 3. A *database program* $T_\Sigma(\bar{x})$ with parameters $\bar{x} = [x_1, \ldots, x_n]$ over relational database schema $\Sigma = \langle \mathcal{T}_\mathcal{L}, \mathcal{R}, IC \rangle$ is an f-term in $\mathcal{L} \cup \mathcal{R}$ whose only free

variables are parameters in $\bar{x}$. A database program is a *transaction* if it is of state sort, otherwise it is a *query*.

Unlike transactions, integrity constraints can be arbitrary situational assertions in our transaction logic. Integrity constraints serve the purposes of defining valid database states and valid transactions. The former type of constraints describes the static semantics of the database and the latter type of constraints describes the dynamic semantics of the database. This distinction leads to the following definition:

Definition 4. Given the relational database schema $\Sigma = \langle \mathcal{T}_{\mathcal{L}}, \mathcal{R}, IC \rangle$, define $p \in IC$ to be a *static constraint* if it is equivalent to an s-formula of the form $(\forall_{state} s)(s::q)$ for some f-formula q. Otherwise it is a *dynamic constraint*.

The transaction logic provides a very powerful tool for the specification of database semantics. To illustrate its expressive power, we may compare it with first-order temporal logic, which is one of the most widely used modal logics for describing computations. In first-order temporal logic[1], there are five modal operators: $\Box, \bigcirc, \Diamond, \mathcal{U}, \mathcal{P}$. For temporal formulas α and β, the informal semantics of the modal operators is:

$\Box\alpha$: from now on α is always true
$\bigcirc\alpha$: α is true in the next state
$\Diamond\alpha$: α is eventually true
$\alpha\mathcal{U}\beta$: α is true until β is true
$\alpha\mathcal{P}\beta$: α precedes β

In applying temporal logic to describe database evolution, the next-state relation and the accessibility relation collapse due to the fact that database evolution graphs are transitive: $\bigcirc\alpha \equiv \Diamond\alpha$ for all α. We may define a mapping δ from the set of formulas in temporal logic to a set of formulas in situational logic, such that a temporal formula α is valid at state s in temporal logic if and only if $\delta(s, \alpha)$ is valid in situational logic.

$\delta(s, \alpha) = s::\alpha$, no temporal operators in α
$\delta(s, \Box\alpha) = (\forall_{state} t)\delta(s;t, \alpha)$
$\delta(s, \bigcirc\alpha) = (\exists_{state} t)\delta(s;t, \alpha)$
$\delta(s, \alpha\mathcal{U}\beta) = (\forall_{state} t)(\delta(s;t, \alpha) \vee$
$(\exists_{state} t_1)(\exists_{state} t_2)(t = t_1;;t_2 \wedge \delta(s;t_1, \beta)))$
$\delta(s, \alpha\mathcal{P}\beta) = (\exists_{state} t)(\delta(s;t, \alpha) \wedge$
$(\forall_{state} t_1)(\forall_{state} t_2)(t = t_1;;t_2 \rightarrow \delta(s;t_1, \neg\beta)))$

The above argument shows that our transaction logic is at least as powerful as first-order temporal logic in specifying dynamic semantics of databases. On the other hand, any integrity constraints in our transaction logic which describe properties of specific transactions are not expressible in first-order temporal logic, because programs are not objects in the formalism. For example, the axioms for the fluent function $modify_n$ in the previous section cannot be expressed with temporal logic. Hence our transaction logic is strictly more expressive than first-order temporal logic, the former can be used to characterize database evolution graphs while the latter is only suitable for specifying properties of Kripke structures.

Now we turn our attention to the semantics of relational databases in the presence of dynamic constraints. Given a database schema $\Sigma = \langle \mathcal{T}_{\mathcal{L}}, \mathcal{R}, IC \rangle$, the *verification* of Σ involves a proof that the theory $\mathcal{T}_{\mathcal{L}} \cup IC$ is consistent, or $\mathcal{T}_{\mathcal{L}} \cup IC$ has a model $\mathcal{M}$ over the language $\mathcal{L} \cup \mathcal{R}$. Since $\mathcal{T}_{\mathcal{L}} \cup IC$ is a theory in first-order logic, schema verification is no more difficult than a first-order consistency problem and taking dynamic constraints into consideration does not increase the complexity of schema verification.

A relational database $DB_\Sigma = \langle \mathcal{T}_{\mathcal{L}}, \mathcal{R}, IC \rangle$ is *valid* if $DB_\Sigma \models IC$. The *validation* of an integrity constraint $\phi \in IC$ against DB_Σ is to show that $DB_\Sigma \models \phi$. However, general constraint validation is made impossible by the fact that many reasonable applications have infinite number of database states. Only a partial model, rather than a complete database DB_Σ, can be maintained for access, which is usually the current state and sometimes a part of database history. We say that an integrity constraint is *checkable* if its validity in the maintained partial model, together with the assumption that the database has been valid in the history, implies its validity in the complete model. For example, if the database only keeps track of the current state, then static constraints are checkable but dynamic constraints are not. If both the current state and the previous state are maintained, then certain transaction constraints become checkable. There has to be certain compromise between the expressiveness of the semantic specification and the ability of the database system to properly maintain the semantics, although a powerful formalism like ours is still useful in expressing the human understanding of the database semantics. In practice, some dynamic constraints can be checked by encoding partial history information into every state. In the next section, we give examples of constraints which are checkable with various amount of history maintained. However a precise characterization of constraint checkability and its relationship with constraint maintainability is out of the scope of this paper.

4. Example Specification

Throughout the section, we draw examples from the employee database below. For notational convenience, we write $select_n(t, i)$ as $f(t)$ where f is the i-th attribute of n-ary tuple t. Also we use the more familiar infix notations for operators such as $+, \in, \subseteq$.

EMP(e-name, e-dept, salary, age, m-status)

DEPT(d-name, chair, location)
PROJ(p-name, t-alloc)
ALLOC(a-emp, a-proj, perc)
SKILL(s-emp, s-no)

Example 1. Suppose our employee database requires that (1) Each employee works for at least one project; (2) Each *alloc* tuple must be associated with a valid project; and (3) No employees should be allocated over 100% of their time. These constraints are part of the static semantics of the database. The following sentences specify them in our transaction logic:

$$(\forall_{state'} s)(\forall_{5tup'} e')(\exists_{2tup'} a')$$
$$(e' \in s{:}EMP$$
$$\rightarrow a' \in s{:}ALLOC \wedge e\text{-}name'(s, e') = a\text{-}emp'(s, a'))$$
$$(\forall_{state'} s)(\forall_{3tup'} a')(\exists_{2tup'} p')$$
$$(a' \in s{:}ALLOC$$
$$\rightarrow p' \in s{:}PROJ \wedge a\text{-}proj'(s, a') = p\text{-}name'(s, p'))$$
$$(\forall_{state'} s)(\forall_{5tup'} e')$$
$$(e' \in s{:}EMP$$
$$\rightarrow sum'(s, \{perc(s, a') | a' \in s{:}ALLOC \wedge$$
$$a\text{-}emp'(s, a') = e\text{-}name'(s, e')\}) \leq 100)$$

Example 2. There is an integrity constraint which says that an employee cannot be single if he was married before. We may represent it as a constraint over states: if an employee in state s_1 is not single and is younger than himself in state s_2, then he cannot be single in s_2.

$$(\forall_{state'} s_1)(\forall_{state'} s_2)(\forall_{5tup} e)$$
$$(s_1{:}e \in s_1{:}EMP \wedge s_2{:}e \in s_2{:}EMP \wedge$$
$$age'(s_1, s_1{:}e) < age'(s_2, s_2{:}e) \wedge m\text{-}status'(s_1, s_1{:}e) \neq S$$
$$\rightarrow m\text{-}status'(s_2, s_2{:}e) \neq S)$$

The above assertion has in fact a built-in assumption that if an employee in s_1 is younger than himself in s_2 then s_2 is reachable from s_1, which is not always true. Two states may very well be in contradition as long as they are not reachable from each other. Hence the right way to specify the constraint is in the form of a transaction constraint which says that if an employee is not single in s_1 and is younger than himself in s_2 in which he is single, then s_2 is not reachable from s_1:

$$(\forall_{state'} s)(\forall_{state} t)(\forall_{5tup} e)$$
$$(s{:}e \in s{:}EMP \wedge s;t{:}e \in s;t{:}EMP \wedge$$
$$age'(s, s{:}e) < age'(s;t, s;t{:}e) \wedge m\text{-}status'(s, s{:}e) \neq S$$
$$\rightarrow m\text{-}status'(s;t, s;t{:}e) \neq S)$$

If employees cannot be rehired then this constraint is checkable with a history of two states: the current state *cs* and the previous state *ps*. Suppose an employee is single in *cs*, then he must be single in *ps*. Since the history before *cs* is valid, the employee is single in any state in the history that can reach *ps* and hence *cs*.

Example 3. As another example of transaction constraints, consider the assertion that an employee retains a skill as soon as he obtains it:

$$(\forall_{state'} s)(\forall_{state} t)(\forall_{5tup} e)(\forall_{2tup} k)$$
$$(s{:}e \in s{:}EMP \wedge s;t{:}e \in s;t{:}EMP \wedge$$
$$s{:}k \in s{:}SKILL \wedge s\text{-}emp'(s, s{:}k) = e\text{-}name'(s, s{:}e)$$
$$\rightarrow s;t{:}k \in s;t{:}SKILL)$$

It is a transaction constraint since if an employee does not possess a type of skill in $s_2 = s_1; t$ which he has in s_1 then s_2 is not reachable from s_1 via a valid transaction. The constraint should not be expressed as saying that the deletion of *skill* tuples is prohibited, because we do want to delete the *skill* tuples associated with an employee when we delete the employee himself. Given that employees are never rehired, this constraint is checkable with a history of two states because the relationship $\subseteq$ is transitive. If the set of skills of an employee in the previous state is a subset of that in the current state, his set of skills in any history state must also be a subset.

The constraint that an employee's salary cannot decrease unless he switches departments is an executability condition of transactions that change the salary attribute:

$$(\forall_{state'} s)(\forall_{state} t)(\forall_{5tup} e)$$
$$(s{:}e \in s{:}EMP \wedge s;t{:}e \in s;t{:}EMP \wedge$$
$$salary'(s, s{:}e) \not\leq salary'(s;t, s;t{:}e)$$
$$\rightarrow (\exists_{state} t_1)(\exists_{state} t_2)(t = t_1;;t_2 \wedge$$
$$e\text{-}dept'(s, s{:}e) \neq e\text{-}dept'(s;t_1, s;t_1{:}e)))$$

The constraint enforces restrictions not only on the two end states of a transaction but also on the possible transitions in between. If a transaction decreases the salary of someone, it has to go through an intermediate state at which the employee switches departments. Thus validity is not determined solely by the two end states. Since $\leq$ is also transitive, this constraint is checkable in three states. If we replace $\leq$ by $\neq$, i.e., the salary of an employee is never the same as before, then the constraint is checkable only with a complete history.

Now we consider transaction constraints in the Structural Model[22] which concern about the reference connection (e.g., employees refer to their departments) and association connection (e.g., allocations are associated with projects). Statically both of these connections are referential constraints, but they have different dynamic behavior: a department should not be deleted if it has employees while all allocations should be deleted along with the deletion of a project.

$$(\forall_{state'} s)(\forall_{3tup} d)(s::(d \in DEPT) \wedge$$
$$\neg(\exists_{5tup'} e')(e' \in s{:}EMP \wedge$$
$$e\text{-}dept'(s, e') = d\text{-}name'(s, s{:}d))$$
$$\rightarrow s;delete_3(d, DEPT)::(d \notin DEPT))$$

$(\forall_{state'} s)(\forall_{state} t)(\forall_{2tup} p)$
$(s{:}p \in s{:}PROJ \wedge s;t{:}p \notin s;t{:}PROJ$
$\rightarrow \neg(\exists_{3tup'} a')(a' \in s;t{:}ALLOC \wedge$
$a\text{-}emp'(s;t, a') = p\text{-}name'(s;t, s;t{:}p)))$

The first constraint serves as a pre-condition for a transaction which deletes *dept* tuples. It is checkable with two states. The second constraint can be subsumed by the referential constraint in Example 1. Therefore dynamically the association connections are equivalent to static referential constraints.

Example 4. There are more complex dynamic constraints which do not fit in the subclass of transaction constraints. They usually involve more states and transactions. Assume our employee application requires that once an employee is fired, he should never be hired again:

$(\forall_{state'} s)(\forall_{state} t_1)(\forall_{5tup} e)$
$(s{:}e \in s{:}EMP \wedge s;t_1{:}e \notin s;t_1{:}EMP$
$\rightarrow \neg(\exists_{state} t_2)(s;t_1;t_2{:}e \in s;t_1;t_2{:}EMP))$

This constraint is not checkable without knowing the complete history of database evolution. To avoid the overhead of history maintenance, we may encode part of the history by having a relation *fire* about those employees fired by the company. Such an encoding makes the constraint statically checkable, by adding a static constraint $(\forall_{state'} s)(\forall_{5tup'} e')(e' \in s{:}FIRE \rightarrow e' \notin s{:}EMP)$. As another example of non-transaction constraints, suppose we require that every transaction is invertible unless it modifies the age of an employee:

$(\forall_{state'} s)(\forall_{state} t_1)$
$((\forall_{5tup} e)(s{:}e \in s{:}EMP \wedge s;t_1{:}e \in s;t_1{:}EMP \wedge$
$age'(s, s{:}e) = age'(s;t_1, s;t_1{:}e))$
$\rightarrow (\exists_{state} t_2)(s = s;t_1;t_2))$

It is not checkable because whenever a transaction is executed, the existence of an inverse transaction needs to be proved. The statement that no projects should last forever is not checkable for the same reason:

$(\forall_{state'} s)(\forall_{2tup} p)$
$(s{:}p \in s{:}PROJ \rightarrow (\exists_{state} t)(s;t{:}p \notin s;t{:}PROJ))$

Example 5. The situational transaction logic we developed can also be used for the specification of transactions. The transaction below cancels a project p, fires those employees who become not involved in any projects, and reduce the salaries of those who still work for some other projects.

transaction *cancel-project*(p, v)
assign$(E, \{a\text{-}emp(a) | a \in ALLOC \wedge$
$a\text{-}proj(a) = p\text{-}name(p)\});;$
foreach $a | a \in ALLOC \wedge a\text{-}proj(a) = p\text{-}name(p)$ **do**
$delete_3(a, ALLOC);;$
$delete_2(p, PROJ);;$
foreach $e | e \in EMP \wedge e\text{-}name(e) \in E$ **do**
if $(\exists_{3tup} a)(a \in ALLOC \wedge a\text{-}emp(a) = e\text{-}name(e))$
then $modify_5(e, 3, salary(e) - v)$
else $delete_5(e, EMP)$
end

Many constraints can also be checked by proving certain properties of the transactions involved, with only a history of one state maintained. This combines model-checking with theorem-proving. For example, the transaction here can be proved to preserve the validity of all transaction constraints in Examples 2 and 3 except that it may violate the one about salary modification if there are employees who work for projects besides p. The validity of the first constraint in Example 4 is also preserved since the transaction does not hire new employees.

Example 6. The previous example illustrates the procedural specification of a transaction. We may also give a declarative specification of the same transaction and rely on a theorem-prover to synthesize a procedural transaction by constructive proofs.

$(\forall_{state'} s)(\exists_{state} t)$
$(s;t{:}p \notin s;t{:}PROJ \wedge$
$(\forall_{5tup} e)(\forall_{3tup} a)(s{:}e \in s{:}EMP \wedge s{:}a \in s{:}ASSIGN \wedge$
$a\text{-}proj'(s, s{:}a) = p\text{-}name'(s, s{:}p) \wedge$
$a\text{-}emp'(s, s{:}a) = e\text{-}name'(s, s{:}e)$
$\rightarrow salary'(s, s{:}e) - v = salary'(s;t, s;t{:}e)))$

The above specification is treated as a theorm. The theorem can be proved and a transaction is constructed as a by-product of the proof. Notice that the deletion of the associated allocations and those employees who do not work for any projects are not specified in the theorem, they are created during the proof to satisfy the integrity constraints in Example 1. Because of the limitation in space, we won't give further description of the proof.

5. Conclusion

We developed a situational transaction logic as the uniform formalism for the specification of the dynamic behavior of the databases, including the dynamic integrity constraints which specify the properties of database state transitions and the transactions whose executions cause the state transitions. Based on the formalism, we defined informally the trade-off between the expressive power of the semantic specification and the validation capability of the database system. Our formalism is very powerful in specifying the dynamic semantics of the database evolution, and yet it remains a first-order logic for which automated theorem-proving techniques are better-understood. We also indicated

the application of our formalism to the verification and synthesis of transactions.

Several directions for future work are possible. We only gave an informal description of constraint checkability. Precise characterizations are needed to control the interaction of specification expressiveness and history maintenance. We may treat checkability as a specification complexity measure and investigate the relationships between various classes of integrity constraints, such as that between static and dynamic constraints[20]. Our formalism is also applicable to the verification and synthesis of transactions. Transaction verification can be combined with constraint validation to make more constraints checkable with less amount of history maintained, which leads to more knowledgable database systems.

Acknowledgement

The first author has benefited from the discussions with Professor Gio Wiederhold, Tom Pressburger, Peter Ladkin, Surajit Chaudhuri, Allen Goldberg, and Arthur Keller. This work was supported in part by the Defense Advanced Research Projects under Contract N39-84-C-0211 for Knowledge Based Management Systems, by Rome Air Development Center under Contract 30602-86-C-0026, by the National Sscience Foundation under Grants DCR-82-14523, and by the Office of Naval Research under Contract N00014-84-C-0706.

References

[1] Abadi, Martin, "Temporal-logic Theorem Proving"; PhD Dissertation, *Tech. Rep.* STAN-CS-87-1151, Dept. Comp. Science, Stanford Univ., Mar.1987.

[2] Abiteboul, S., Vianu, V., "A Transaction Language Complete for Database Update and Specification"; *Proc. 6th ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, 1987, 260-268.

[3] Brodie, M., "On Modelling Behavioural Semantics of Data"; *Proc. 7th Int'l Conf. VLDB*, 1981, 32-42.

[4] Burstall, R., "Formal Description of Program Structure and Semantics in First-order Logic"; *Machine Intelligence 5*, B. Meltzer and D. Michie (editor), Edinburgh University Press, Edinburgh, Scotland, 1969, 79-98.

[5] Casanova, M., Bernstein, P., "A Formal System for Reasoning about Programs Accessing a Relational Database"; *ACM Trans. Programming Languages and Systems*, Vol.2, No.3, July 1980, 386-414.

[6] Castilho, J., Casanova, M., Furtado, A., "A Temporal Framework for Database Specifications"; *Proc. 8th Int'l Conf. VLDB*, 1982, 280-291.

[7] Clifford, J., Warren, D., "Formal Semantics for Time in Databases"; *ACM Trans. on Database Systems*, Vol.8, No.2, June 1983, 214-254.

[8] Ehrich, H., Lipeck, U., Gogolla, M., "Specification, Semantics, and Enforcement of Dynamic Database Constraints"; *Proc. 10th Int'l Conf. VLDB*, 1984, 301-308.

[9] Gardarin, G., Melkanoff, M., "Proving Consistency of Database Transactions"; *Proc. 5th Int'l Conf. VLDB*, 1979, 291-298.

[10] Green, C., "Application of Theorem Proving to Problem Solving"; *Proc.IJCAI*, Washington, D.C., May 1969, 219-239.

[11] Kung, C., "On Verification of Database Temporal Constraints"; *Proc. ACM SIGMOD Conf.*, 1985, 169-179.

[12] Manchanda, S., Warren, D., "Towards a Logical Theory of Database Updates"; *Tech. Rep.* 86/19, Dept. Computer Science, State University of New York at Stony Brook, Jul.1986.

[13] Manna, Z., Waldinger, R., "A Deductive Approach to Program Synthesis"; *ACM Trans. Programming Languages and Systems*, Vol.2, No.1, Jan.1980, 90-121.

[14] Manna, Z., Waldinger, R., "Problematic Features of Programming Languages: A Situational-logic Approach"; *Acta Informatica*, Vol.16, 1981, 371-426.

[15] Manna, Z., Waldinger, R., "How to Clear a Block: A Theory of Plans"; *Journal of Automated Reasoning*, Vol.3, No.4, Dec.1987, 343-377.

[16] Manna, Z., Waldinger, R., "The Deductive Synthesis of Imperative LISP Programs"; *Proc. AAAI*, Seattle, Aug.1987, 155-160.

[17] Manna, Z., Waldinger, R., *The Logical Basis for Computer Programming, Vol.2: Deductive Techniques*, Addison-Wesley, to be published.

[18] McCarthy, J., "Situations, Actions, and Causal Laws"; *Semantic Information Processing*, M. Minsky (editor), MIT Press, Cambridge, Mass., 1968, 410-417.

[19] Sheard, T., Stemple, D., "Automatic Verification of Database Transaction Safety"; *Coins Tech. Report* 86-30, Univ. Massachusetts at Amherst, 1986.

[20] Vianu, V., "Dynamic Constraints and Database Evolution", *Proc. 2nd ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, 1983.

[21] Waldinger, R., Lee, R., "PROW: A Step Toward Automatic Program Writing"; *Proc. IJCAI*, Washington, D.C., May 1969, 241-252.

[22] Wiederhold, G. and ElMasri, R., "The Structural Model for Database Design"; *Entity-Relationship Approach to System Analysis and Design*, P. Chen (editor), North Holland, Amsterdam, 1980.