PERGAMON

Expert Systems with Applications 19 (2000) 133–142

Expert Systems with Applications

www.elsevier.com/locate/eswa

EDTTDT: an expert design tool for temporal database transaction

M. Qutaishat*

Department of Computer Science, University of Jordan, Amman, Jordan

Abstract

This paper describes an expert tool for temporal database transaction design, EDTTDT, developed to assist transaction designers in creating efficient, safe and reliable transactions. When applied to a temporal database design EDTTDT first derives specific Knowledge about the application. It then accepts a user-designed transaction, converts it into its internal form (And/Or tree) and performs the following processing on the transaction: optimization, which attempts to improve the efficiency of the transaction; temporal safety verification, which checks whether the transaction preserves the consistency of the database; amendment, which corrects any errors detected during safety verification; and temporal analysis, which attempts to detect the semantic errors that may exist in the transaction. The resulting transaction is safe, and may have been improved with respect to efficiency and reliability if it was not originally well designed.

Keywords: Expert design tool; Temporal database; Temporal integrity constraints; Temporal knowledge; Database transaction

1. Introduction

1.1. Temporal aspects

A database contains data that, together with its associated applications, models a real world enterprise. Temporal information is associated with data and data processing in many applications: database systems, planning and scheduling systems, diagnosing and monitoring systems, information systems, and real time systems. In many of such database applications, temporal information has to be stored, retrieved and related to other temporal information. Hence, research areas in temporal database environment have been classified into three categories (Maiocchi & Pernici, 1992; Tansel, 1993): conceptual data-modeling, whose focus is time representation; database systems concerned with management of temporal information; and artificial intelligence, to address time reasoning.

Three possible approaches for the development of temporal databases have been suggested (Maiocchi & Pernici, 1992). The first is called rollback and extends the semantics of the relational model to incorporate time directly by storing all the past states of the database. The relation can be regarded as a sequence of snapshot relations indexed by transaction time. On the other hand, database that supports valid time is termed historical databases, i.e. the time in which the stored information corresponds to reality. The valid time of an event is the wall clock time at which the event occurred in the modeled reality. However, the term temporal database is used to describe the database that support both valid and transaction time. While a rollback database views tuples as being valid at sometimes as of that time, the historical databases always view tuples as being valid at some moment as now. Tuples in temporal databases can be viewed as being valid at some moment relative to some other moment, completely capturing the history of retroactive/proactive changes.

1.2. Integrity aspects

The importance of integrity in database management has been increasingly recognized in recent years (Casanova, Tucherman & Furtado, 1988; Chomicki & Toman, 1995; Dimitris, 1993; Hsu & Imielinski, 1985; Qian, 1988; Sandhu, 1994; Wang, Fiddian & Gray, 1991). Yet there is no consensus on what is meant by integrity. However, integrity constraints remain an essential part of every database application. From a classification point of view, integrity constraints can usually be divided into two categories (Chomicki & Toman, 1995): static (referring to a static snapshot or current state of the database) and temporal (referring to a sequence of snapshots, ordered in time, i.e. past and future state in addition to the current state). From a definition point of view, there have been four definitions of data integrity reported in the literature (Sandhu, 1994). The most general of these, also reported in Courtney (1989), is based on the concept of expectation of data quality: data has

* Tel.: +962-6-5355000, ext. 2901.
E-mail address: maq@ju.edu.jo (M. Qutaishat).

0957-4174/00/$ - see front matter
PII: S0957-4174(00)00028-2

integrity to the extent that its quality meets, or exceeds, the quality requirements that users expect of it. This definition incorporates durability requirements, whereas the others only address safety requirements. The second and third definitions are both based on the ability to modify data. One of these, discussed in Jajodia and Sandhu (1990) defines the scope of integrity to be safeguards against the faulty modification of data. The other narrows the scope further to be safeguards against the unauthorized modification of data. The fourth definition considers integrity as one-directional information flow in a lattice (Courtney, 1989; Sandhu, 1994).

The integrity constraints of a database are semantic rules derived from the real world enterprise. They capture semantic information that the user (designer) possesses about the enterprise. They describe either the properties that the database state must obey or the allowable changes in the database state. The former type of integrity constraints is referred to as a static constraint and the latter as a temporal/transition constraint. Examples of these types of integrity constraints are:

- Employee's age is less than 70 (static constraint);
- Age of an employee should never decrease (transition constraint).

A database state is said to be consistent if it satisfies all the integrity constraints of the database. The constraints are usually enforced by checking whether they are true after every database change. Temporal constraints require checking not only the current database state but also the entire database history. This is, however, clearly a complex task. We propose a semi-automatic approach to derive the historical information that has to be kept in a database store. The historical information is stored as a set of extra triggers and rules in every database state to form the so-called incremental integrity. The incremental integrity (Dimitris, 1993) checking method which is adopted in EDTTDT is based on the premise that constraints are known to be satisfied prior to the database transaction and are to be verified after the update, specially those that are affected by it. Incremental integrity checking is formulated by specializing integrity constraints with respect to the anticipated types of updates and by performing simplification on the specialized forms.

1.3. Motivations and design goals

The design of database transactions in a temporal database environment is a difficult task, as it requires both general knowledge about temporal database systems and transaction design, and specific knowledge about a particular temporal database application. The former type of knowledge includes general knowledge relevant to the design of transactions, for instance, what properties a transaction has, e.g. is it a logical unit of work (Elmasri & Navathe, 2000), and various useful techniques that help in the design of good quality transactions, e.g. how to derive cheaper integrity checks. The latter type of knowledge is concerned with a particular database. This includes, for instance, the entities and the temporal integrity constraints in the database, and the actions to be taken when certain integrity constraints are violated. Temporal constraints allow imposing restrictions on the transactions over the database like:

"Age of an employee cannot decrease"

Both types of knowledge are desirable in the design of transactions. Designers who lack either or both types of knowledge may come up with transactions that contain various types of faults. The common faults include transactions that are inefficient, unsafe, or their effects are not what the designers have in mind, i.e. they are unreliable. In addition, even though a designer possesses both types of knowledge, ensuring the safety of transaction is still a difficult and error-prone task, as the number of integrity constraints in database is likely to be large. In addition, time handling requirements include various issues such as time representation, temporal reasoning, temporal knowledge acquisition, and the management of temporal properties. Therefore, there is a need for automatic design tools to assist transaction designers. It is this need that motivated the development of EDTTDT, an expert design tool for temporal database transaction.

EDTTDT's main design goal is to help transaction designers to design efficient, safe and reliable transactions.

This paper focuses on the various types of knowledge EDTTDT contains and its implementation using the Prolog language. The rest of the paper is organized as follows. In Section 2, we give both an architectural and a functional overview of EDTTDT. In Section 3, we briefly discuss the contents of EDTTDT's knowledge base. In Section 4, we describe the implementation of a prototype EDTTDT using Prolog. In Section 5, we draw the conclusions.

2. An overview of EDTTDT

2.1. Preliminaries

EDTTDT has been developed in the context of temporal relational databases, where a temporal relational database is viewed as Past Fragment of Temporal Logic (Past First Order Temporal Logic with temporal operations referring solely to the past) (Chomicki & Toman, 1995; Tansel, 1993). Database integrity constraints are expressed as function-free and range-restricted Horn-clauses (with equality) in the language of the database theory (McCune & Henschen, 1989; Nicolas, 1982). Both the integrity constraints and the database state before a transaction is applied are assumed to be consistent.

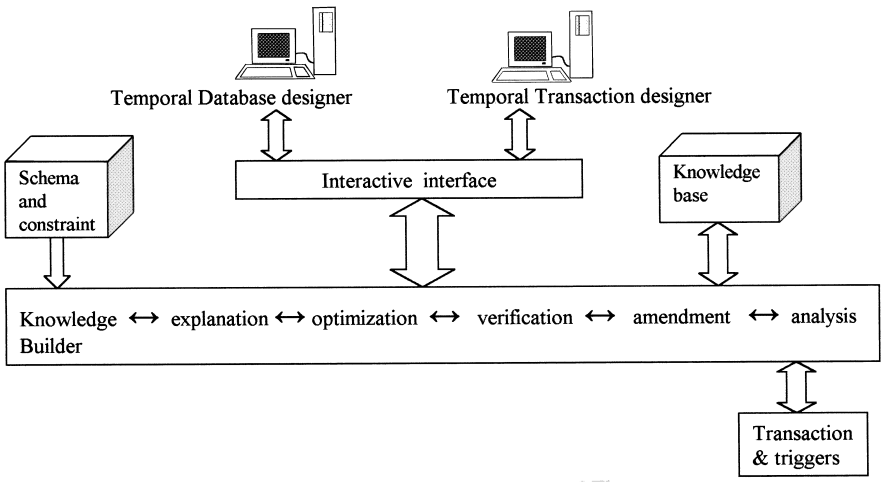

Fig. 1. Architectural organization of EDTTDT.

The three classes of database update operations (insert/delete/modify) are assumed to be different in their complexity (and thus in their execution costs). Here, the increasing order of complexity defined in Abiteboul and Vianu (1984), i.e. insert, delete and modify, is assumed to apply.

Update operations in a transaction are classified into two types, namely primary operations and secondary operations. A secondary operation in a transaction is one that is required either by a primary operation or by another secondary operation for the purpose of preserving the consistency of the database. Those operations in a transaction that are not secondary operations are primary operations. Therefore, a secondary operation is only executed if its primary operation has an effect on the database state. This is because a primary operation which has no effect on the database state will not violate any temporal integrity constraints and thus does not need any secondary operations.

Database transactions are assumed to be composed of three sections: the temporal parameter section, which contains all the parameters used by preconditions and operations; the temporal precondition section which contains conditions that ensure that all the primary operations of the transaction, if successfully executed, have effects on the database; and the temporal transaction body, which consists of one or more update operations possibly connected using if–then–else control constructs.

2.2. An architectural overview of EDTTDT

EDTTDT resembles an expert system with a limited explanation/advice facility. It provides a menu interface to its end-users (who are either database or transaction designers). Fig. 1 shows an architectural overview of EDTTDT, and identifies its major components whose names reflect their functions.

2.3. A functional overview of EDTTDT

The processing performed by EDTTDT when it is applied to a database application consists of two stages, namely the *specific knowledge building stage* and the *transaction processing stage*. The first stage is only performed once for each database application (although the specific knowledge base may undergo changes), while the second stage is performed whenever a transaction is submitted for processing. These two stages are summarized below:

(a) *The specific knowledge building stage:*
(a.1) Build database schema facts;
(a.2) Build temporal integrity constraint facts;
(a.3) Build temporal database consistency maintaining rules (TDCM-rules).

(b) *The transaction processing stage:*
(b.1) Accept a temporal database transaction;
(b.2) Optimize the transaction body;
(b.3) Verify the temporal safety of the transaction body. If any unsafe or semantically unsafe situation is detected, call the transaction amendment procedure;
(b.4) Once again optimize the transaction body, as step (b.3) may introduce new redundancies;
(b.5) Analyze the parameters, attempting to detect possibly omitted temporal preconditions or update operations;
(b.6) Analyze the transaction body, attempting to detect potential temporal semantic errors;
(b.7) Check whether all the temporal preconditions required by the primary operations exist. If any is missing, call the transaction amendment procedure;
(b.8) Optimize the temporal preconditions;
(b.9) Analyze the temporal preconditions, attempting to detect potential temporal semantic errors.

All the above steps are assisted by the explanation/advice facility that has been built into each end-user menu. It either explains the situation that a transaction designer is facing or generates tailored advice regarding a specific situation. The resulting transaction is safe, and may have been improved with respect to efficiency and reliability if it was not originally well designed.

A partial working session (simplified for exposition purposes) is presented below:

1. The system starts by reading a database schema from the file "dbfile" and converting the schema into internal form. The schema of the database is as follows:

   Create entity student [sid:integer; sname:string; placed:Boolean,derived] key is [sid]
   student [ValidTimeStart, ValidTimeEnd]

   Create entity major [jid:integer; jname:string] key is [jid]
   major [ValidTimeStart, ValidTimeEnd]

   Create entity department [did:integer;…] key is [did]
   department [ValidTimeStart, ValidTimeEnd]

   Create entity registration [rid:integer;…]…
   registration [ValidTimeStart, ValidTimeEnd]
   etc.

2. The system is now reading temporal integrity constraints from the file "integrity_file" and converting them into their internal form. The database may contains the following integrity constraints:

   Contains(student.sid, registration.sid)
   Contains(department.did, registration.did)
   Contains(major.jid, registration.jid)
   All s in student: is_redundant_with (s.registered, in (s.sid, registration.sid))
   etc.

   Operation:
   Delete [a, b, c] from student
   Will violate the constraint:
   Contains(student.sid, registration.sid)
   If the following condition evaluates false after the operation:
   Not registration(a,_8,_9,_10) (where _N represents any value)
   What is the action to be taken if the constraint is violated?
   1—Reject 2—Propagate 3—Designer control
   4—Advice 5—Explanation
3. The system is now building the temporal database consistency rules (TDCM-rules) with the involvement of the database designer.
4. The system then performs the following processing on the transaction: optimization, which attempts to improve the efficiency of transaction; temporal safety verification, which checks whether the transaction preserves the consistency of the database; amendment, which corrects any errors detected during safety verification; and temporal analysis, which attempts to detect the semantic errors that may exist in the transaction.
5. The resulting transaction is safe, and may have been improved with respect to efficiency and reliability if it was not originally well designed.

   Transaction drop-student (drop-st, dept, st-major)
   Precondition
   Registration(drop-st, dept, st-major);
   Begin
   Delete [drop-st, _20, _21, _22] from registration;
   If offering(dept, st-major, _23)
   Then modify offering from [dept, st-major, _24] to [dept, st-major, total + 1]
   Else insert [dept, st-major, 1] into offering;
   Modify student from [drop-st, _25, true] to [drop-st, _25, false];
   If department(dept, _26, _27) and major(st-major, _28)
   Then commit
   Else abort;
   End

3. The knowledge base of EDTTDT

The knowledge base of EDTTDT is composed of two constituent knowledge bases, namely the *specific knowledge base* and the *general knowledge base*.

3.1. The specific knowledge base

The specific knowledge base for temporal database application is built, with the involvement of the database designer and incorporated into EDTTDT. The specific knowledge base contains both structural and temporal behavioral information about the application. The former is concerned with the entities and integrity constraints of the database, while the latter is concerned with the allowable operations on the relations in the database and the relationships between these operations.

3.1.1. Structural information: schema and temporal integrity constraints

The structural information includes:

- the entities in the database;
- the attributes of these entities and their features, e.g. ValidTimeStart, ValidTimeEnd, keys or derived attributes;
- the relationships between these entities;
- the temporal integrity constraints as specified in high-level specification languages; and
- the internal forms of the temporal integrity constraints.

The general forms of the schema definition statements are as follows:

define entity Entity-Name [(Attribute: construct);]
key is [(Attribute);]
define relationship Relationship-Name [(Attribute: construct);]
key is [(Attribute);]

A set of temporal integrity constructs specifications. For example, three of these constructs are as follows:

- Containment: contains(set, subset)
- Universal quantification: all t in R { where p(b)}:Q(t)
- Past First Order Temporal Logic (Past FOTL)
  $C :\neg (\exists x, c_1, c_2, m)(reserved(c_1, m) \wedge$
  reserved$(c_2, m) \wedge$
  (valid_time(c_1)OVERLAP valid_time(c_2)))

3.1.2. Temporal behavioral information: temporal database consistency maintaining rules

Although any database update operation is physically executable on the database, not all of them are meaningful in the real world enterprise of which the database is a model, and those which are meaningful are often constrained in certain ways to preserve the consistency of the database. The allowable update operations on the database, the properties of the database (the integrity constraints) that must be preserved by these operations, and the actions to be taken when certain properties are violated are all captured in the temporal database consistency maintaining rules (TDCM-rules). The TDCM rules are in effect the physical and operational interpretations of the semantics of the database application. The temporal queries and integrity constraints can be expressed using recently proposed Past First Order Temporal Logic (Past FOTL) (Chomicki & Toman, 1995; Henschen, McCune & Naqvi, 1984; Tansel, 1993). The expressions of Past FOTL use the following notation:

- A finite set of relation symbols.
- A finite set of constant symbols.
- An infinite set of variables.
- Equality and inequality symbols ($=, \neq, >, <, \geq, \leq$).
- The logical connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$).
- Quantifiers ($\forall$, $\exists$).
- Past-tense temporal connectives:
  ● ("previous time"), and since.
- Other well known temporal connectives:
  ■ ("always in the past"),
  ◆ ("sometime in the past") can be defined using since:
  ◆ A = true since A
  ■ A = $\neg$ ◆ $\neg$ A.

Other Past FOTL formulas are defined by the following rules: if A, B are formulas and x is a variable, the following are formulas:

$A \wedge B$, $A \vee B$, $\varnothing A$, $A \Rightarrow B$, $\exists \times A$, $\forall \times A$, ●A, and A since B.

A formula is called

- *pure first-order* if it does not use temporal connective at all;
- *temporal* if it is of the form ●A, and A since B (i.e. the main connective is temporal).

3.2. The general knowledge base

The general knowledge base contains techniques, which are concerned with specific knowledge building, transaction optimization, transaction safety verification, unsafe transaction amendment and safe transaction analysis. In addition, it contains a certain amount of general knowledge concerning database system and transaction design, which is used by the advice/explanation facility to generate advice or explanation as requested by users. This consists of both information similar to what we would find in a database, and the constructs that organize and provide relationships among these facts. New knowledge can also be created from inferences based on relationships among facts. Because representing time is a key concern in developing EDTTDT, the choice of methods for representing temporal knowledge (Stuart & Peter, 1995; Susan & Gregory, 1997) have been governed by:

1—the ability to define and use time in various knowledge base constructs;
2—the ability to represent and maintain temporal relationships;
3—the ability to define varying temporal granularities; and
4—the ability to represent absolute or relative time.

The various techniques incorporated into EDTTDT have been chosen to overcome or to alleviate the gravity of the various types of faults that are commonly found in the temporal database transactions. They are either modifications or extensions of those developed by other researchers (Abiteboul & Vianu, 1984; Brodie & Ridjanovic, 1984; Chakravarthy, Grant & Minker, 1990; Dimitris, 1993; Gardarin & Melkanoff, 1979; Henschen et al., 1984; King, 1981; McCune & Hencshen, 1989; Shaerd & Stemple, 1985, 1989; Stemple, Mazumdar & Sheard, 1987; Wang et al., 1991; Yao, 1979), in addition to those mentioned in Section 1, or new techniques developed by us. The procedures that implement these techniques are given in Table 1, together with brief descriptions of their function. In what follows, we briefly discuss the origins and the purpose of these procedures.

The TDCM-rule_building_procedure is based on a

Table 1
Procedures contained in EDTTDT's general knowledge base

Procedure	Function
Specific knowledge building procedures:	
Schema_fact_building_procedure	Read and convert database schema into internal form
Costraint_fact_building_procedure	Read and convert temporal integrity constraints into internal form
TDCM-rule_building_procedure	Build the TDCM-rules of the database with the involvement of the database designer
Transaction_normalization_procedure	Read and convert transaction into their internal forms, which are And/Or trees (normalization)
Temporal transaction optimization procedures:	
Temporal_Semantic_precondition_optimization_procedure	Remove redundant preconditions using temporal semantic information in the form of the temporal integrity constraints without affecting the truth value of the preconditions conjunction
Temporal_Semantic_operation_optimization_procedure	Remove redundant operation or replace expensive operation with cheaper but equivalent ones using temporal semantic information in the form of the temporal integrity constraints
Temporal_Syntactic_precondition_optimization_procedure	Remove redundant preconditions using syntactic simplification rules activated on the preconditions and attempts to remove those which are (syntactically) subsumed by the others. It is based on a set of rules similar to the following simplification rule: $(f1 \wedge f2) \wedge (f1 \rightarrow f2) \Leftrightarrow f1$ (where f1 and f2 are arbitrary formulas)
Temporal_Syntactic_operation_optimization_procedure	Remove redundant operation or replace expensive operations with cheaper but equivalent ones using syntactic simplification rules based on a set of pre-proved rewriting rules, which state the relationships and interactions between pairs of update operations from both the same and different classes
Temporal_Transaction_safety_amendment_procedure	Verify whether a transaction is safe and call the amendment procedure if it is not based on a modified version of a standard theorem prover, which is coupled with heuristics for cutting down both the number and complexity of theorems (integrity constraints) to be proved in the new database
Temporal_Transaction_amendment_procedure	Amend unsafe or semantically unsafe situations detected during transaction safety verification. It consists of five amendment rules
Temporal transaction analysis procedures:	
Temporal_Parameter_analysis_procedure	Detect missing precondition or update operation
Temporal_Precondition_analysis_procedure	Detect preconditions which are not required by any primary operations
Temporal_Operation_analysis_procedure	Detect potential temporal semantic errors, e.g. several independent transactions have been put within one transaction

procedure proposed in McCune and Hencshen (1989), called "Procedure B", which takes a constraint I and a class of update operation U, and either proves that U cannot violate I, or produces a formula F which is a simplified instance of I. The difference is that our TDCM-rule_building_procedure produces simplified integrity constraints to be tested after an update operation is applied rather than before, as with those produced by McCune's "Procedure B". The TDCM-rule_building_ procedure is also more efficient than McCune's "Procedure B", as it is based upon syntactic criteria, while McCune's "Procedure B" is based upon the resolution principle (McCune & Hencshen, 1989).

The four transaction optimization procedures can be classified into two groups, namely the semantic optimization group and the syntactic optimization group. The latter implements our syntactic optimization technique, which is closely related to the technique reported in Abiteboul and Vianu (1984), but different from it in that ours is based on a set of pre-proved rewriting rules which states the relationships and interaction between pairs of update operation from both the same and different classes.

The transaction_safety_verification_procedure resembles a theorem prover, which is related to the verification techniques reported in Gardarin and Melkanoff (1979), McCune and Hencshen (1989) and Shaerd and Stemple (1985). However, it is different from those as it has been designed as a modified version of the standard theorem prover.

The transaction amendment procedure is related to the work of Stemple et al. (1987) in which attempts are to amend an unsafe transaction by adding either integrity checks or compensating operations. However, our amendment procedure is more user friendly.

The transaction analysis procedure is similar to Stemple et al. (1987) in which attempts are made to generate some of the effects of a transaction, e.g. the changes in cardinality of the updated relation, so that transaction designers can be alerted if some unexpected result appears.

4. EDTTDT implementation

4.1. The use of Prolog

A prototype EDTTDT has been implemented on a Sun workstation using the Prolog programming language. The

system code amounts to some 12,000-source lines. Prolog was chosen because it has been characterized and demonstrated by other researchers as an appropriate language for defining and implementing knowledge-based systems or expert systems in general (Gallaire, Minker & Nicolas, 1984; Holsapple & Whinston, 1995; Storey & Goldstein, 1988).

In the following subsection, some of the functions of the various procedures listed in Table 1 are discussed in more detail and the techniques for implementing these procedures are highlighted.

4.2. The specific knowledge building procedures

4.2.1. Grammar rule based procedures

The *schema_fact_building_procedure*, the *constraint_fact_building_procedure* and the *transaction_normalization_procedure* have a common feature: they all read in some input of restricted forms and then convert the input into suitable internal forms. These three procedures can easily be implemented using Prolog grammar rules that state the mapping between the original forms and their internal forms.

For instance, the two top level grammar that rule the *schema_fact_building_procedure* are as follows:

```
schema ([Name, relation, Attributes, VTS, VTE, Key])
–> [create, entity], entity_name (Name), [ '[' ],
attribute_list (Attributes), [ ']' ], [key, is], [ '[' ],
key_list (Key), [ ']' ].

schema ([Name, relationship, Attributes, VTS, VTE,
Key]) –> [create, relationship], relationship_name
(Name), [ '[' ], attribute_list (Attributes), [ ']' ],
[key, is], [ '[' ], key_list (Key), [ ']' ].
```

The database schema is internally represented using facts of the following format:

```
FramRelation (Relation_name, Relation_type,
Attributes_list, VTS, VTE, Key)
```

where Relation_name is the name of a relation, Relation_type either takes the value "entity" if the relation is an entity relation, or "relationship" if it is relationship relation, Attributes_list contains the attributes of the relation and their domains, VTS (ValidTimeStart) and VTE (ValidTimeEnd) denote the time period during which a fact is true with respect to the real world, and Key contains the key of the relation. The integrity constraints are internally represented using facts of the following format:

```
integrity (Integrity_Id, Integrity_form, Constraint_type,
Clausal_form, Status)
```

where Integrity_Id is a unique identifier of a constraint, Integrity_form is the constraint in its original form, Constraint_type represents the type of the constraint, Clausal_form is a list of literals representing the clausal form of the constraint, and Status is a list of values, either "in" or "not_in", where an "in" value indicates that the corresponding literal in Clausal_form will appear in the simplified form of the constraint.

A database transaction is internally represented using an And/Or tree constructed from the temporal preconditions and operations in the transaction. A transaction And/Or tree is represented using a collection of facts. The root is represented using a fact of the following format:

```
temporal_transaction (Tran_name, Hierarchy_list)
```

where Tran_name is the name of the transaction and Hierarchy_list contains the keys of the hierarchy of the root.

Both the preconditions and the operations are uniformly represented using facts of the following format:

```
hierarchy (key, Cond_type, Cond_body, Hierarchy_list)
```

where

```
Key is a unique identifier of this fact;
Cond_type is one of:
   precondition, postcondition, condition, operation or
   intermediate_Cond;
If Cond_type = precondition/postcondition/condition
   Then Cond_type is the precondition/postcondition/
   condition itself;
If Cond_type = operation
   Then Cond_type is the operation itself
If Cond_type = intermediate
   Then Cond_type is either "and" or "or";
```

Hierarchy_list is either an empty list, indicating no hierarchy, or a list containing the keys of the hierarchy of the fact.

4.2.2. Transaction_normalization_procedure

The input transaction may be arbitrarily complex, depending on the facilities provided by the language. It is the goal of normalization to transform the query to a normalized form to facilitate further processing. With temporal relational languages such as TSQL (Tansel, 1993), the most important transformation is that of the query qualification (e.g. WHERE, AFTER, BEFORE, etc.) which may be an arbitrary complex, quantifier-free predicate, preceded by all necessary quantification ($\forall$ or $\exists$). For the theorems that follow, assume that A, B, and C are valid-time relations. The transformation of the quantifier-free predicate is straightforward in Prolog using the equivalence rules for the logical operations ($\wedge$, $\vee$, $\neg$, π, σ, $\cup$, and $\times$) as shown in Table 2.

4.2.3. TDCM-rule_building_procedure

The main problem involved with the temporal constraints is that of temporal inference. For example, the simplest

Table 2
Equivalence rules

$A \wedge B \Leftrightarrow B \wedge A$	$A \vee B \Leftrightarrow B \vee A$
$A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C$	$A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C$
$A \wedge (B \vee C) \Leftrightarrow$	$A \vee (B \wedge C) \Leftrightarrow$
$(A \wedge B) \vee (A \wedge C)$	$(A \vee B) \wedge (A \vee C)$
$\neg (A \wedge B) \Leftrightarrow \neg A \vee \neg B$	$\neg (A \vee B) \Leftrightarrow \neg A \wedge \neg B$
$\neg (\neg A) \Leftrightarrow A$	$A \cup B \Leftrightarrow B \cup A$
$A \times B \Leftrightarrow B \times A$	$\sigma p1(\sigma p2(A)) \Leftrightarrow \sigma p2(\sigma p1(A))$
$A \cup (B \cup C) \Leftrightarrow (A \cup B) \cup C$	$A \times (B \times C) \Leftrightarrow (A \times B) \times C$
$A \times (B \cup C) \Leftrightarrow$	$\sigma p(A \cup B) \Leftrightarrow \sigma p(A) \cup \sigma p(B)$
$(A \times B) \cup (A \times C)$	
$\sigma p(A - B) \Leftrightarrow \sigma p(A) - \sigma p(B)$	$\pi \times (A \cup B) \Leftrightarrow$
	$\pi \times (A) \cup \pi \times (B)$

relation between intervals is *meet*. The function *Duration* gives the difference between the ValidTimeEnd VTE and the ValidTimeStart VTS:

$\forall$t Interval (t) $\Rightarrow$ Duration (t) = (Time (VTE(t)) − Time (VTS (t))

Two intervals *meet* if the end time of the first equals the start time of the second. It is possible to define predicates such as *Before*, *After*, *During*, and *Overlap* solely in terms of *meet*, but it is more intuitive to define them in terms of points on the time scale:

T1 = Interval 1, t2 = Interval 2, t3 = Interval 3
$\forall$t1, t2 Meet (t1,t2) $\Leftrightarrow$ Time (VTE (t1)) = Time (VTS (t2))
$\forall$t1, t2 Before (t1,t2) $\Leftrightarrow$ Time (VTE (t1)) < Time (VTS (t2))
$\forall$t1, t2 After (t1,t2) $\Leftrightarrow$ Before (t1,t2)
$\forall$t1, t2 During (t1,t2) $\Leftrightarrow$ Time (VTS (t2) $\leq$ Time (VTS (t1) $\wedge$ Time (VTE (t1)) $\leq$ Time (VTE (t2))
$\forall$t1, t2 Overlap (i,j) $\Leftrightarrow$ $\exists$t3 During (t3,t1) $\wedge$ During (t3,t2).

The system builds the TDCM-rules of the database with the involvement of the database designer. Using Past FOTL notations, constraints can be formulated over the database like:

(a) "A student should not register in a course for which he did not study its pre-requisites previously unless he registers them in the same semester in which the course will be taken" is expressed as a Past FOTL formula as:

$$C :\neg (\exists x, c_1, c_2, r_1)(student(x) \wedge \text{register}(x, c_1) \wedge \text{requisite}(c_1, r_1) \wedge \neg \bullet \text{register}(x, r_1) \wedge \neg (\text{valid\_time}(c_1)\text{EQUALS valid\_time}(r_1)))$$

where x is a variable that refer to student, c1 is a variable that refers to a course, and r1 is a variable that refers to a pre-requisite of c1. This format can be easily translated into Prolog rules.

Table 3
Valid formulas

$A \vee \text{false} \Leftrightarrow A$	$A \wedge \text{false} \Leftrightarrow \text{false}$
$A \wedge \text{true} \Leftrightarrow A$	$A \vee \text{true} \Leftrightarrow \text{true}$
If $A \Rightarrow B$, then $A \wedge B \Leftrightarrow A$	If $A \Rightarrow B$, then $A \vee B \Leftrightarrow B$
$A \wedge (A \vee B) \Leftrightarrow A$	$\forall x A(x) \Rightarrow \exists x A(x)$
$\forall x A(x) \Rightarrow A(y)$	$A(y) \Rightarrow \exists x A(x)$
$\forall x(x \neq y \vee A(x)) \Leftrightarrow A(y)$	$\exists x(x = y \wedge A(x)) \Leftrightarrow A(y)$
$\forall x(A(x) \wedge B(x)) \Leftrightarrow$	$\exists x(A(x) \vee B(x)) \Leftrightarrow$
$\forall x A(x) \wedge \forall x B(x)$	$\exists x A(x) \vee \exists x B(x)$
$\forall x(A(x) \vee B) \Leftrightarrow$	$\exists x(A(x) \wedge B) \Leftrightarrow$
$\forall x A(x) \vee B$ where x $\neg$ in B	$\exists x A(x) \wedge B$ where x $\neg$ in B

(b) "In a hotel reservation system no room should be reserved for one customer that are to be reserved at the same time by another customer" is expressed as:

$$C :\neg (\exists x, c_1, c_2, m)(reserved(c_1, m) \wedge \text{reserved}(c_2, m) \wedge (\text{valid\_time}(c_1)\text{OVERLAP valid\_time}(c_2)))$$

where c1, c2 are both variables that refer to customers and m is a variable that refers to a room.
(c) "No resource is used by more than one activity in any time" is expressed as:

$$C :\neg \exists(a_1, a_2, \ldots a_n)(\ldots(used\_resource(O_1, a_1, RS_1) \wedge used\_resource(O_2, a_2, RS_2) \ldots \wedge (\text{valid\_time}(a_1)\text{OVERLAP valid\_time}(a_2) \ldots \text{OVERLAP valid\_time}(a_n)).$$

where a_1, a_2, a_n are variables that refer to activities, $O_1, O_2, \ldots, O_n$ are variables that refer to objects, and $RS_1, RS_2, \ldots, RS_n$ are variables that refer to the required resources.

4.2.4. Elimination of redundancy

Temporal relational languages can be used uniformly for syntactic and semantic data control. In particular, a transaction may be enriched with several temporal predicates to achieve relation correspondence and ensure semantic integrity:

◆ A $\Leftrightarrow$ true since A
■ A $\Leftrightarrow$ $\neg$ ◆ $\neg$ A

The enriched query quantification may then contain redundant predicates. Such redundancy may be eliminated by simplifying the qualification with the valid formulas given in Table 3.

These formulas are concerned with: how to detect update operations which cannot violate a certain integrity constraint; how to derive simplified instances of integrity constraints based on syntactic criteria; and how to derive compensating operations from simplified instances of integrity constraints. A complex operation has a meaning derived from the meaning of its parts. Each connective can be thought of as a function. Here, truth tables can be used

Table 4
Truth table showing validity options

A	B	A ∨ B	(A ∨ B)∧ ¬ B	((A ∨ B)∧ ¬ B) ⇒ A
F	F	F	F	T
F	T	T	F	T
T	F	T	T	T
T	T	T	F	T

not only to define the connectives, but also to test valid operations. For each operation, we can calculate the truth valve of the entire transaction. This can be done by building a set of truth tables for the application (Premises ⇒ Conclusion) and checking all the applicable operations, as can be seen in Table 4.

These formulas have been encoded into a specific knowledge (declarative procedures). These procedures test all possible ways that an integrity constraint may be violated by update operations acting on relations occurring in it. If a class of update operations U may violate a constraint C, a formula F will be produced which is either a simplified instance of C or C itself. Then the database designer (who presumably is responsible for building TDCM-rules) is involved in choosing an action A to be taken when C is false. Finally, a TDCM-rule of the following format will be built and inserted into the specific knowledge base of EDTTDT:

TDCM_rule (Rule_Id, C, U, F, A, Rest)

where Rule_Id represents a unique identifier of this TDCM-rule, and Rest represents other possible compensating actions which can be used to maintain the consistency of the database.

5. Conclusions

EDTTDT, an Expert design tool for temporal database transaction, has been described in this paper. Its knowledge based and implementation in Prolog have been discussed. The importance of this research lies in both the establishment of the feasibility of integrating, using the knowledge-based system technology, a comprehensive set of techniques into a useful transaction design tool, and in the demonstration of the benefits such a design tool can bring. The major contributions of this research lie in the various types of procedures that comprise the general knowledge base of this design tool.

The future potential applications of EDTTDT have also been considered. Apart from being used as a database transaction design tool, EDTTDT can also be applied, with little or no modification, in several other areas including Transaction Design Teaching System, Database Design Aid, and Integrity Constraint Interpreter.

References

Abiteboul, S. & Vianu, V. (1984). Transaction in relational databases (preliminary report). *Proceedings of the 10th VLDB Conference* (pp. 46–56).

Brodie, M. L., & Ridjanovic, D. (1984). On the design and specification of database transactions. In M. L. Brodie, J. Mylopulos & T. W. Schmidt, *On conceptual modeling* (pp. 277–306). Berlin: Springer.

Casanova, M.A., Tucherman, L. & Furtado, A.L. (1988). Enforcing inclusion dependencies and referential integrity. *Proceedings of the 14th VLDB Conference* (pp. 38–49).

Chakravarthy, S., Grant, J., & Minker, J. (1990). Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, *15* (2), 162–207.

Chomicki, J., & Toman, D. (1995). Implementing temporal integrity constraints using an active DBMS. *IEEE Transactions on Knowledge and Data Engineering*, *7* (4), 566–582.

Courtney, R. (1989). Some informal comments about integrity and the integrity workshop. *Proceedings of the International Workshop on Data Integrity*. National Institute of Standards and Technology, Special Publication.

Dimitris, P. (1993). Integrity constraint and rule maintenance in temporal deductive knowledge bases. *Proceedings of the 19th VLDB Conference* (pp. 146–158).

Elmasri, R., & Navathe, S. (2000). *Fundamentals of database systems*, Reading, MA: Addison-Wesley.

Gallaire, H., Minker, J., & Nicolas, J-M. (1984). Logic and database: a deductive approach. *ACM Computing Surveys*, *16* (2), 153–185.

Gardarin, G. & Melkanoff, M. (1979). Proving consistency of database transactions. *Proceedings of the Fifth VLDB Conference* (pp. 291–298).

Henschen, L. J., McCune, W. W., & Naqvi, S. A. (1984). Compiling constraint checking programs from first-order formulas. In H. Gallaier, J. Minker & J-M. Nicolas, *Advances in database theory* (pp. 145–196), vol. 2. New York: Plenum Press.

Holsapple, C., & Whinston, A. (1995). *Business expert systems*, Homewood, IL: Irwin.

Hsu, T. & Imielinski, T. (1985). Integrity checking for multiple updates. *Proceedings of the ACM SIGMOD Conference* (pp. 152–168).

Jajodia, S. & Sandhu, R. (1990). Polyinstantiation integrity in multilevel relations revisited. *DBSec* (pp. 297–308).

King, J.J. (1981). QUIST: a system for semantic query optimization in relational database. *Proceedings of the Seventh VLDB Conference* (pp. 510–517).

Maiocchi, R., & Pernici, B. (1992). Automatic deduction of temporal information. *ACM Transactions on Database Systems*, *17* (4), 647–688.

McCune, W. W., & Hencshen, L. J. (1989). Maintaining state constraints in relational databases: a proof theoretic basis. *ACM Transactions on Database Systems*, *36* (1), 46–68.

Nicolas, J-M. (1982). Logic for improving integrity checking in relational databases. *Acta Informatica*, *18* (3), 227–253.

Qian, X. (1988). An effective method for integrity constraint simplification. *Proceedings of the 4th IEEE on Data Engineering Conference*, (pp. 338–345).

Sandhu, R. (1994). *On five definitions of data integrity*. *Database security*, vol. VII(A-47). Amsterdam: North-Holland.

Shaerd, T. & Stemple, D. (1985). Coping with complexity in automated reasoning about database systems. *Proceedings of the ACM SIGMOD Conference* (pp. 246–369).

Shaerd, T., & Stemple, D. (1989). Automatic verification of database transaction safety. *ACM Transactions on Databases Systems*, *14* (3), 322–369.

Stemple, D., Mazumdar, S. & Sheard, T. (1987). On the modes and meaning of feedback to transaction designers. *Proceedings of the ACM SIGMOD Conference* (pp. 374–386).

Storey, V. C., & Goldstein, R. C. (1988). A methodology for creating user

views in database design. *ACM Transactions on Database Systems, 13* (3), 305–338.

Stuart, R., & Peter, N. (1995). *Artificial intelligence—a modern approach*, Englewood Cliffs, NJ: Prentice-Hall.

Susan, C., & Gregory, M. (1997). A framework for developing and evaluating expert systems for temporal business applications. *Expert Systems Applications, 12* (3), 393–404.

Tansel, A. (1993). *Temporal databases, design, and implementation*, Menlo Park, CA: Benjamin/Cummings.

Wang, X., Fiddian, N. & Gray, W. (1991). The development of a knowledge-based database transaction design assistant. *Proceedings of Database and Expert Systems Applications* (pp. 356–361). Berlin: Springer.

Yao, S. (1979). Optimization of query evaluation algorithms. *ACM Transactions on Database Systems, 4* (2), 133–155.