TALE — A Temporal Active Language and Execution Model

Avigdor Gal*, Opher Etzion**[1] and Arie Segev***[2]

[1] Department of Information Systems Engineering,
Faculty of Industrial Engineering and Management,
Technion-Israel Institute of Technology,
Haifa, 32000,
Israel

[2] Haas School of Business, University of California and
Information & Computing Sciences Division,
Lawrence Berkeley Laboratory Berkeley,
CA 94720, USA

Abstract. Complex applications in domains such as decision support systems and real time systems require a functionality that is achieved by combining the active and temporal database technologies. In this paper we present TALE, a Temporal Active Language and Execution model. TALE is a temporal active database programming language, combined with an execution model that enables a correct and efficient processing of operations. As such, TALE is a step in accommodating software engineering challenges in modern information systems. TALE primitives are presented using examples and an EBNF. The run-time control mechanism of the model is introduced and TALE properties, namely active and temporal capabilities, and reflective programming capabilities are discussed.

Keywords: Active databases, Temporal databases, Information modeling, Database programming languages

---

* The work was conducted while the author was in the Technion. He is currently at the Department of Computer Science, University of Toronto, Toronto, Ontario, M5S 3H5 CANADA.

** The work of this author was supported by the fund for the promotion of research at the Technion.

*** The work of this author was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

1 Introduction

Complex applications in domains such as decision support and real time systems require the processing of temporal data and the execution of programs as a response to events. For example, in decision support applications the validity of current decisions may be dependent upon data that was valid in the past. Consequently, these decisions should be re-evaluated whenever there are retroactive changes of data. A temporal active database [9] provides the required support for those applications by combining the functionality of both temporal databases [16] and active databases [3]. A temporal active database supports time stamped data, and contains a mechanism to activate operations in response to detected events. This combination of the temporal and the active capabilities enables the execution of direct updates, derived updates and retrieval operations that refer to past, present and future time points.

In this paper we introduce TALE, a Temporal Active Language and Execution model. TALE is an extension of the model presented in [9], [19], in which rules that affect past or future valid times were briefly discussed. The model consists of a high level language[3] that enable the use of *events, conditions* and *actions* relating to past, present and future time points. An execution model translates the definition language into a *dependency graph,* an executable data structure that directs the correct and efficient execution of transactions.

The main contribution of this paper is the execution model that provides a wide range of temporal and active inter-relationships. The need for such a model was recognized during the 1993 ARPA/NSF Workshop on an infrastructure for temporal databases. However, only limited discussions of the temporal and active inter-relationships exist in both research areas. Several works (e.g. [3] and [14]) in the active database research area discuss the use of temporal events in active databases. Since these works do not utilize temporal data models, temporal conditions and temporal actions are disabled. Temporal models (e.g. [17], [22], [23] and [20]) discuss the propagation of the temporal effect on a reactive activation of programs, while TALE enables the use of reflective programming [7] as well. Temporal query languages (e.g. [21]) enable the manipulation of time attributes as part of the language, and the use of defaults in defining time bounded values. However, none of the existing temporal query languages offers the wide range of possible inter-relationships between a valid time of a deriving value and a valid time of a derived value that is supported in TALE. Dependency graphs are used in several active data models (e.g. [1] and [7]). In [1], the dependency graph is generated for analysis purposes only, using a given set of ECA rules. TALE and PARDES [7] use the dependency graph as an executable data structure, with a well defined translation mechanism from the user definition of operation clauses to the graph. Both [1] and [7] lack the capability of handling temporal conditions and temporal actions.

As a concrete motivating case study, we present a simple example of a traffic law enforcement system, that imposes a fine as a penalty for speeding offenses.

[3] The support of the language for constraints was discussed in [13].

If a fine is not paid within 30 days of the traffic violation, the fine is doubled. If a fine is not paid within 60 days of the traffic violation, a court order to summon the driver is requested. In addition to the fines, a demerit point system is used. A driver that violates the speed limit in a residential area receives 2 demerit points that are valid for a period of two years. A driver that violates the speed limit in roads other than residential roads receives 3 demerit points that are valid for a period of three years. A driver that acquires 6 demerit points or more must pass a traffic school course; otherwise, the driver's driving license is revoked until the driver passes the course, at which point the driver gets a new driving license.

Basic concepts of the underlying data model are defined in Section 2. Section 3 presents the language primitives, and the execution model of TALE is given in Section 4. Section 5 presents the properties of TALE, including the temporal and active capabilities of the model. The EBNF definition of the language's grammar is presented in Appendix A.

2 The temporal database

The underlying temporal data model is a bi-temporal database [16] with a valid time (t_v) which designates the time points at which the value is considered to be true in the modeled reality, and a transaction time (t_x) which designates the time when a value becomes current in the database. In this paper, we focus on the manipulation and use of valid times, while assuming an automatic assignment of a transaction time to each new value. We use a generic data model, in which an *object* is generically defined as an instance of a class or a tuple in a relation, and a *property* is defined as an attribute in the object-oriented model and a column in the relational model. The term *class* is used to define either a class in the object-oriented model, or a relation in the relational model.

Figure 1 presents a partial schema definition that consists of the name and the properties of three classes: *Driver, Traffic-Citation* and *Speed-Violations* of the traffic law enforcement system. *Driver* includes seven properties that are relevant to each driver. *Traffic-Violations* is a set property where each member of the set (called *Violation*) is a reference to an object in the *Traffic-Citation* class. *Points* is the number of demerit points associated with the driver, and *Traffic-Course-Deadline* is the deadline for participating in a traffic course before the driver's license is revoked. *Driving-License?* is a boolean property (with a domain of "true" and "false" values) that represents the status of the driver's driving license, and *Civil-Service-Employee?* is a boolean property that identifies the driver as a civil service employee. It is worth noting that although Figure 1 consists of a compound property (*Traffic-Violations*), it can be easily translated into a relational data model, as well as into an object-oriented data model.

Points, Traffic-Course-Deadline and *Driving-License* are derived properties. Derived properties, characterized using *(derived)* in Figure 1, are those properties whose value is derived from the values of other properties. The exact definition of their derivation formula is given in Section 3.

Class = *Driver*
Properties = *Driving-License-Number*
Name
Traffic-Violations: set of
Violation (reference to Traffic-Citation)
Points (derived)
Traffic-Course-Deadline (derived)
Driving-License? (derived)
Civil-Service-Employee?

Class = *Traffic-Citation*
Properties = *Citation#*
Driving-License-Number
Date
Violation (reference to Speed-Violations)
Violating-Speed
Fine-Due (derived)
Paid?
Court-Order-Request (derived)
Violation-Points (derived)

Class = *Speed-Violations*
Properties = *Violation-Code*
Road-type
Speed-Limit
Road-Points
Maximal-Fine

Fig. 1. Schema definition of the traffic law enforcement system

Traffic-Citation contains nine properties that are relevant to a single traffic violation: *Citation#*—an identifier of the specific traffic violation, *Driving-License-Number*—the violating driver's driving license, *Date*—the date of the violation, *Violation-Code*—a reference to an object in the *Speed-Violations* class, *Violating-Speed*—the violating speed, *Fine-Due*—the amount of the due fine, *Paid?*—a boolean property that record whether a fine is paid or not, *Court-Order-Request*—the date when a court order should be requested, and *Violation-Points*—the number of demerit points due for the violation.

Speed-Violations stores the general details of violations in different types of roads. It consists of five properties: *Violation-Code*—the code of the violation type, *Road-Type*—the type of the road, *Speed-Limit*—the legitimate speed limit, *Road-Points*—the number of demerit points that a driver obtains as a result of being caught violating the speed limit of the specified road, and *Maximal-Fine*—the maximal possible fine limit.

A *variable* is an instance of a property that is associated with a specific

Points=

2,	*Dec* 1 1994,	[*Dec* 1 1994, *Dec* 1 1996)
5,	*Nov* 12 1995,	[*Nov* 12 1995, *Dec* 1 1996)
3,	*Nov* 12 1995,	[*Dec* 1 1996, *Nov* 12 1998)
	t_x	t_v

Fig. 2. Values of John Doe's *Points*

object. A variable in a temporal database consists of a set of values, such that each value has its own time stamp. For example, Figure 2 illustrates the changes to the variable *Points* of the driver John Doe. The variable *Points* of John Doe has overlapping values ("2" and "5") during [Nov 12 1995, Dec 1 1996), since during that time interval the variable consists of demerit points that relate to two different violations. A selection criterion that chooses the value(s) that is (are) valid in an interval for which a variable has overlapping values can be based on several preference relations. As a default, we assume that a value val_j is preferred to a value val_i iff $t_x(val_j) > t_x(val_i)$.[4] It is worth noting that in this figure, and in the rest of the paper, a single day granularity is assumed.

An *event* is an instantaneous occurrence that can be detected by the database's event detector. It can be the result of a user-initiated signal operation (e.g. the conclusion of a trial), a sensor's output (e.g. the date of requesting a court order as signaled by the system clock), or an update operation (e.g. a change in a *Speed-Limit*). A thorough discussion of events' classification and detection techniques can be found in [2] and [5].

A *user update* operation in a temporal active database updates a single object in a given temporal element. For example, an object can be *inserted* to the database or *deleted* from it. Object's variables are modified as part of an update operation using a *modify* operation, or as part of an *insert* operation.

3 TALE — the language primitives

An *operation* is an executable entity that is activated as a result of an event detection, either conditionally or unconditionally. The set of operations in a temporal active database are partitioned according to their level of interaction with the database variables. *User update* operations are operations that update the database but do not retrieve it. *Apply* operations are operations that possibly retrieve the database but do not update it. There are two types of operations that can retrieve and update the database, namely the *enforce* operation and the *derive* operation. In this section we present the language primitives of TALE

[4] In the temporal model as presented in [10], the t_x is replaced with a *decision time*, a time type that captures the time of the real-world event(s) that lead to the transactions that affect the temporal database.

using examples. The EBNF definition of the language's grammar is given in Appendix A.

TALE language primitives define *operation clauses*, a behavioral description of operations. Operation clauses are defined in a declarative fashion, with the possible use of auxiliary programs, when the execution logic of a desired operation is too complex to be easily represented in a declarative form, or when the database uses existing programs (e.g. *legacy systems*).

Temporal relationships are specified through the use of *participants*, a valid time bound properties of the form $\langle$property-name, $t_v\rangle$. A property name followed by a t_v defines the values that are retrieved by the operation, or the bounded effect of the operation on the generation of new values.

An operation clause is composed of four components: an *expression* that defines the execution logic of the operation, a *when* clause that denotes a precondition for activation, a *triggered-by* clause that consists of events that trigger the operation, and a *signals* clause that consists of events that are signaled by the operation. If the *triggered-by* statement is omitted, an implicit triggering is assumed, such that any change to a variable that is part of the operation clause activates the operation. For example, consider the following example:

Driving-License?:= *Renew-License(Driving-License-Number, Name)*
when not(*Driving-License?*)
triggered-by Pass-Traffic-Course

The expression consists of an auxiliary program *Renew-License* that updates an instance of the property *Driving-License?*. The operation is triggered whenever the event *Pass-Driving-Course* occurs, and is activated if the value of *Driving-License?* is false. If the *triggered-by* clause is omitted in this example, then any change of an instance of *Driving-License-Number, Name* or *Driving-License?* triggers the operation. Hence, the following two examples carry the same semantics:

(a) *Driving-License?*:= *Renew-License(Driving-License-Number, Name)*
when not(*Driving-License?*)

(b) *Driving-License?*:= *Renew-License(Driving-License-Number, Name)*
when not(*Driving-License?*)
triggered-by Driving-License-Number, Name, Driving-License?

Several operation clauses can be joined to a single *compound operation*, using the form, as follows.

exp_1 *when* $cond_1$
[*triggered-by* $ev_{11}, \ldots, ev_{1i}$]
[*signals* $ev_{21}, \ldots, ev_{2j}$];
...;
exp_i *when* $cond_i$ | *otherwise*
[*triggered-by* $ev_{31}, \ldots, ev_{3m}$]
[*signals* $ev_{41}, \ldots, ev_{4n}$]

The operation clauses are assumed to have a total order priority according to their location in the compound operation. Thus, $cond_2$ is interpreted as $cond_2 \wedge \neg cond_1$. The *otherwise* element is interpreted as $\neg cond_{i-1} \wedge \neg cond_{i-2} \wedge \ldots \wedge \neg cond_1$. The following sections consist of the description of the different types of operations, namely enforce, derive and apply.

3.1 Enforce

An enforce operation evaluates the truth value of a boolean expression on the database state. The primitives that are used by the enforce operation consist of auxiliary programs, participants, constants, binary predicates (e.g. $=$, $>$) and logical connectors (e.g. $\wedge$, $\vee$, $\neg$). For example, the operation clause *Fine-Due* ≤ 500 limits a fine to be less than \$500. Although it is equivalent to the first order logic formula $\forall x \in$ *Traffic-Citation* : *Fine-Due*$(x) \leq 500$, the former representation is much easier to write and comprehend [11].

A constraint violation results in invoking a *stabilizer* [8], an alternative operation activated in the wake of a constraint violation to restore the database's consistency. The trivial, and most conservative stabilizer is *abort*, according to which a transaction that violates the constraint fails. An alternative mechanism for the example given above is the *repair transaction* which changes the input transaction's value to be exactly \$500.

When the participants of an operation clause are not of the same class, a *matching* process, similar to a join operator in relational algebra, is used to identify the objects that are involved in the operation. In this paper we use an implicit matching, based on [6].

3.2 Derive

A derive operation preserves a data dependency by deriving the values of variables. For example, *Fine-Due* $:=$ 15 * (*Violating-Speed* $-$ *Speed-Limit*) defines an operation that derives the fine that is due for a given violation.

While an enforce operation clause (usually) defines merely a dependency that should be preserved, a derive operation also infers update operations required to restore the dependency. The right hand side of a derive operation clause contains a formula for maintaining the correct values of the elements in the left hand side. The derivation expression has the following general form.

$$\text{derivative} := \text{assignment [valid-in } \tau\text{]}$$

The derivation expression includes the *valid-in* argument which denotes the valid time of the derived values. As a unique property of TALE, it is possible to specify an explicitly defined valid time for a derived value that is not necessarily related to the valid times of the deriving values. If the valid time is not specified, then it is derived from the temporal characteristics of the deriving values, using default derivation rules. For example, consider the derive operation clause $x := 5 * b$, and suppose that x and b are properties of the same class. If the value

of $\alpha.b$ changes during [Sep 1 93, Sep 3 93), then the new value of $\alpha.x$ would be valid during [Sep 1 93, Sep 3 93) as well. However, in $x := 5 * \langle b, t\rangle$ *valid-in* t+6, if $\alpha.b$ changes during [Sep 1 93, Sep 3 93), then the new value of $\alpha.x$ would be valid during [Sep 7 93, Sep 9 93). In the latter example, t is an argument bound to the valid time of b.

A special case of the derive operation is when the database is updated by an auxiliary program. In this case the derivative may not be unique and is written as a sequence $\langle derivative_1, \tau_1\rangle, \ldots, \langle derivative_n, \tau_n\rangle$.

3.3 Apply

An *apply* operation calls an auxiliary program that is applied by the database, but does not affect it. For example, an existing software for planning traffic school's class schedules uses the knowledge that exists in the traffic law enforcement system, yet it does not affect it. Hence, the following apply operation clause is available in the system.

apply Plan-Schedule (Driving-License-Number, Name, Driving-Course-Deadline)
triggered-by Traffic-Course-Deadline

The properties that the auxiliary program requires for its operation appear in the parenthesis. It is worth noting that operations that do not interact with the database can also be specified using the apply operation clause. Such an operation takes the following form.

apply auxiliary-program *triggered-by* $\text{ev}_1, \ldots, ev_n$

3.4 An example

Figure 3 presents an exemplary set of operation clauses for the case study. (o_1) computes the fine that is the result of a single traffic violation as a function of the violating speed and the valid time of the violation. According to (o_1), the fine is 15 times the difference between the legal speed and the actual speed during the first month, and is doubled during the second month. The fine balance is reduced to 0 when the fine is paid. The approach taken in (o_1) updates the database proactively, as if the temporal event has already occurred. This approach can assist in answering queries of the form: according to the knowledge that is available today, what would be the fine that is due in 80 days from today? This approach is especially helpful in decision support systems, where decisions that affect the future are based on assumptions that are known today. An alternative approach is to update the database only when the event has actually occurred, using an event *Double-Fine* that signals whenever now=t+31 for some instance. According to this approach, operation (o_1) would be:

(o_1) *Fine-Due* := 15 * (⟨*Violating-Speed*, t⟩ − *Speed-Limit*) *valid-in* [t,t+31)
30 * (⟨*Violating-Speed*, t⟩ − *Speed-Limit*) *valid-in* [t+31, t+61)
triggered-by Double-Fine
0 valid-in [now, ∞) when *Paid?*

```
(o1) Fine-Due                    :=
       15 * (⟨Violating-Speed, t⟩ − Speed-Limit) valid-in [t,t+31)
       30 * (⟨Violating-Speed, t⟩ − Speed-Limit) valid-in [t+31, t+61)
       0                         valid-in [now, ∞)    when Paid?
(o2) Fine-Due                    ≤ Maximal-Fine
       stabilizer=(Fine-Due:=Maximal-Fine)
(o3) Court-Order-Request         :=
       null                          when Paid?
       ts(Violating-Speed)+61        otherwise
(o4) Violation-Points            :=
       0                         valid-in [now, ∞)
                                 when Violation-Points ≠0
                                 triggered-by Pass-Traffic-Course
       Road-Points               valid-in [t, t+2years)
                                 when Road-type=residential
       Road-Points               valid-in [t, t+3years)
                                 otherwise
(o5) Points                      := sum(Violation-Points)
(o6) Traffic-Course-Deadline     :=
       null                          when Points < 6
       ts(Points)+30                 when not(Civil-Service-Employee?)
       ts(Points)+60                 otherwise
(o7) Driving-License?            :=
       Renew-License(Driving-License-Number, Name)
                                 when not(Driving-License?)
                                 triggered-by Pass-Traffic-Course
       false                     valid-in [now, ∞)
                                 when now=Traffic-Course-Deadline
                                 triggered-by Nullify-Driving-License,
                                      Traffic-Course-Deadline
(o8) Apply Plan-Schedule (Driver.Driving-License-Number, Name,
                                      Traffic-Course-Deadline)
                                 triggered-by Traffic-Course-Deadline
```

Fig. 3. Operations of the traffic law enforcement system

(o_2) limits the maximal fine to be paid. In case of a violation, the *Fine-Due* variable receives the maximal possible fine, using a repair transaction stabilizer. (o_3) computes the date for a court order to be requested. This date is nullified if the payment of the fine was confirmed. The *Violation-Points* are computed in (o_4). If the driver passes a traffic school course, then all of the driver's previous points are erased. Otherwise, the number of the points and their validity is defined according to the type of road on which the violation occurred. The driver's total points are calculated in (o_5) as the sum of points of all violations. Due to the bounded temporal effect of operations, the sum of points accumulates with respect to their validity, as presented in Figure 2.

(o_6) computes the traffic course deadline. A driver that gained 6 demerit points or more should participate in a traffic course. t_s represents "the starting time point of a valid time." If a driver has less than 6 points, then the *Driving-Course-Deadline* receives a "null" value. If a driver has more then 6 points (total order priority assumption) and the driver is not a civil service employee, then the traffic school deadline is set to one month after the first time the driver has passed the 6 points limit. If a driver has more then 6 points and the driver is a civil service employee (total order priority assumption), then the traffic school deadline is set to two months after the first time the driver has passed the 6 points limit. (o_7) computes the validity of the driving license, using the auxiliary program *Renew-License*. (o_8) applies the *Plan-Schedule* program, triggered by changes to *Traffic-Course-Deadline* variables.

4 From definitions to executable entities

In this section we present the basic manipulation of operation clauses to generate executable entities (Section 4.1), and a temporal dependency graph (Section 4.2), which monitors the execution process of a temporal active database (Section 4.3).

4.1 The relationships of operations with database elements

Operations are generated from a given set of operation clauses (o_1)-(o_n). Compound operation clauses are decomposed to basic operation clauses, and the relationships among database elements are determined. We define the relationships of an operation o with database elements by using the following four sets.

UPDATE-SET(o): a set of pairs $\langle p, t_v \rangle$, where p is a property that can be updated as a result of the activation of o and t_v is a temporal element during which p is updated.

TRIGGER-SET(o): a set of pairs $\langle p, t_v \rangle$, where p is a property, an operation, or an event that can activate o, and t_v restricts the temporal range in which o is triggered. For example, if p is a property, then a modification of p during t_v triggers o.

REQUEST-SET(o): a set of pairs $\langle p, t_v \rangle$, where p is a property that can be retrieved as a result of the activation of o. By definition, if p is in the request set of o, than p cannot trigger an operation; thus, TRIGGER-SET(o) $\cap$ REQUEST-SET(o) $= \emptyset$.

SIGNAL-SET(o): a set of events that are signaled as a result of the activation of o. For example, in a warehouse inventory system, the activation of an operation for supplying a product may result in signaling a *Low-Inventory* event. A central order system is signaled by this event and issue, as a result, an order for the product. Since events are the means of notifying the external world on occurrences within the application, we assume that events are not further related with other operations. Therefore, there are no implicit relationships among operations due to *signals* clauses.

These sets are directly inferred from the linguistic constructs, and no additional information is required. For example, $x := \langle a, t_{v2} \rangle + \langle b, t_{v3} \rangle$ *valid-in* t_{v1} is a derive operation that infers the following relationships sets:

- UPDATE-SET(o)=$\{\langle x, t_{v1} \rangle\}$.
- TRIGGER-SET(o)=$\{\langle a, t_{v2} \rangle, \langle b, t_{v3} \rangle\}$, due to the implicit triggering property.
- REQUEST-SET(o)=SIGNAL-SET(o)=$\emptyset$.

These relationships are interpreted as follows. Assume that x, a and b are properties of the same class. If the value of $\alpha.a$ is modified in t_{v2}, then the value of $\alpha.b$ in t_{v3} is retrieved, and the value of $\alpha.x$ in t_{v1} is derived and updated.

According to the sets definitions, an operation clause with an explicit *triggered-by* clause imply that the REQUEST-SET consists of all the participants that do not trigger the operation. For example, in $x := \langle a, t_{v2} \rangle + \langle b, t_{v3} \rangle$ *valid-in* t_{v1} *triggered-by* a, TRIGGER-SET(o)=$\{\langle a, t_{v2} \rangle\}$, and REQUEST-SET($o$)=$\{\langle b, t_{v3} \rangle\}$ ($\{\langle a, t_{v2} \rangle, \langle b, t_{v3} \rangle\}$−TRIGGER-SET($o$)).

Table 1 presents the relationship sets of the operations of the case study. The SIGNAL-SET in this example equals $\emptyset$ for each operation; thus, it is eliminated from Table 1. Each operation has a two-digit number, the first of which defines the original operation-clause.

Paid? is included in the TRIGGER-SET of operation (o'_{32}), although it does not appear explicitly in the second clause of operation (o_3) (Figure 3). This is a result of the negation of the condition of the first operation. The same applies for operations (o'_{42}),(o'_{43}) (*Violation-Points*), and (o'_{62}),(o'_{63}) (*Points*). The properties that participate in the matching process are added to the TRIGGER-SET of operations (o'_{11}), (o'_{12}), (o'_{21}), (o''_{21}), (o'_{42}), (o'_{43}), and (o'_{51}). Operation (o'_{21}) appears in the TRIGGER-SET of the stabilizer (o''_{21}), since the stabilizer is (conditionally) activated by the operation.

4.2 The temporal dependency graph

The control of a transaction at runtime is based on a *temporal dependency graph*, an executable data structure that monitors the activation of operations to achieve consistency in a minimal number of update operations and using maximal parallelism [10]. The temporal dependency graph extends previous works in active databases [15], [7], [1] to include the temporal functionality. It consists of all the relationships among properties, events and operations in a database. The temporal dependency graph is generated from a given database structure, a given set of operation clauses, and a given set of events. The operation clauses are translated into operations, and the relationship sets are defined for each operation (Section 4.1).

The temporal dependency graph TDG = (V, E) is a directed graph. The set of nodes V designate the application's elements, properties, events and operations. The set of edges E denote relationships through which the application elements interact. There are three types of edges, *data edges*, *trigger edges*, and *request*

Operation number	Update-Set	Trigger-Set	Request-Set
(o'_{11})	{(Fine-Due, [t, t+31))}	{(Violating-Speed, t), Speed-Limit, Speed-Violations.Violation-Code, Traffic-Citation.Violation-Code}	∅
(o'_{12})	{(Fine-Due, [t+31, t+61))}	{(Violating-Speed, t), Speed-Limit Speed-Violations.Violation-Code, Traffic-Citation.Violation-Code}	∅
(o'_{13})	{(Fine-Due, [now, ∞))}	{Paid?}	∅
(o'_{21})	∅	{Fine-Due, Maximal-Fine, Speed-Violations.Violation-Code, Traffic-Citation.Violation-Code}	∅
(o''_{21})	{Fine-Due}	(o'_{21})	{Maximal-Fine, Speed-Violations.Violation-Code, Traffic-Citation.Violation-Code}
(o'_{31})	{Court-Order-Request}	{Paid?}	∅
(o'_{32})	{Court-Order-Request}	{ Violating-Speed, Paid?}	∅
(o'_{41})	{(Violation-Points, [now, ∞))}	{Pass-Traffic-Course}	{Violation-Points}
(o'_{42})	{(Violation-Points, [t, t+2year))}	{Road-Points, Violation-Points, Speed-Violations.Violation-Code, Traffic-Citation.Violation-Code, Road-Type}	∅
(o'_{43})	{(Violation-Points, [t, t+3year))}	{Road-Points, Violation-Points, Speed-Violations.Violation-Code, Traffic-Citation.Violation-Code, Road-Type}	∅
(o'_{51})	{Points}	{Violation-Points, Driver.Driving-License-Number Traffic-Citation.Driving-License-Number}	∅
(o'_{61})	{Traffic-Course-Deadline}	{Points}	∅
(o'_{62})	{Traffic-Course-Deadline}	{Points, Civil-Service-Employee?}	∅
(o'_{63})	{Traffic-Course-Deadline}	{Points, Civil-Service-Employee?}	∅
(o'_{71})	{Driving-License?}	{ Pass-Traffic-Course}	{Driving-License?, Name, Driving-License-Number}
(o'_{72})	{Nullify-Driving-License,	{Traffic-Course-Deadline} Traffic-Course-Deadline}	{Driving-License?}
(o'_{81})	∅	{Traffic-Course-Deadline}	{Driver.Driving-License-Number, Name}

Table 1. The relationship sets of the case study

edges. Data edges connect properties in the relationship sets with operations, and enable the transfer of data values from variables to operations, and vice versa. A valid time temporal element can be associated with a data edge, to designate the edge's temporal range of effect. A temporal element τ that is associated with a data edge (p, o) restricts the retrieval of any variable $\alpha.p$ to τ. A temporal element τ that is associated with a data edge (o, p) restricts the update of any variable $\alpha.p$ to τ. This association enables reasoning about the temporal relationships in

the database, which cannot be done in other dependency graph based models. However, the temporal dependency graph can be reduced to a dependency graph in the non temporal case by omitting the time restrictions of data edges.

Trigger edges connect events with operations and constraints with their associated stabilizers. A trigger edge between two operations o_i and o_j, that do not origin from the same operation is generated if exists a property that is in the UPDATE-SET(o_i) and in the TRIGGER-SET(o_j), with a possible temporal overlapping. For example, *Points* is in the update set of operations (o_{51}), and in the trigger set of operation (o_{61}), (o_{62}), and (o_{63}) (Table 1). A trigger edge is generated between operation (o_{51}), and each of the operations (o_{61}), (o_{62}) and (o_{63}), since there are no temporal limitations associated with these edges. In similar, a request edge between two operations o_i and o_j that do not origin from the same operation is generated if exists a property that is in the UPDATE-SET(o_i) and in the REQUEST-SET(o_j), with a possible temporal overlapping.

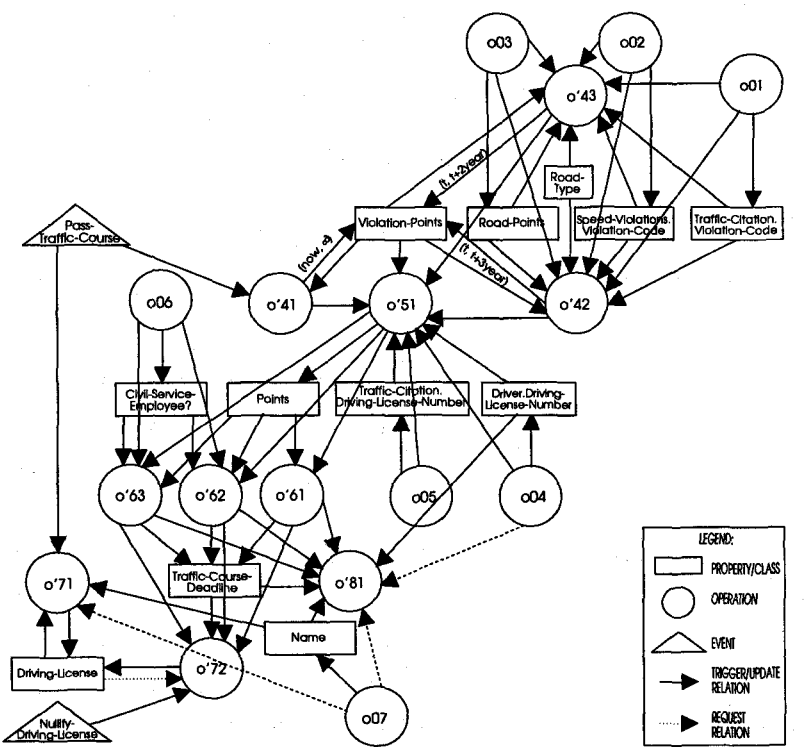

Fig. 4. A partial temporal dependency graph of the traffic law enforcement system

Figure 4 presents a partial example of the temporal dependency graph, with operation clauses (o4)-(o8) of Figure 3. Temporal elements are associated with

data edges and represent the effect of operations in a given temporal element. Trigger edges between operations represent the triggering of an operation as a result of the activation of one of its predecessors. request edges are designated by dotted edges. Operations o_{01}-o_{07} are user update operations, designating the update of a single variable. For example, o_{03} updates the *Road-Points* variable.

The time complexity of generating the temporal dependency graph is computed in [10]. Its worse case is bounded by $O(|V|^3)$, the time complexity of generating the trigger and request edges that are not explicitly given by the relationships sets. The number of nodes and edges reflects the number of schema-elements and the relationships among them, and not the number of objects represented in the database. Thus, the space complexity of the graph, which is bounded by $O(|V| + |E|)$ is much smaller than the size of the data stored in the database. It is worth noting that the generation of the graph is a pre-processing phase, carried out only whenever there is a change in the database structure, either changes to static schema definitions or changes in operation clauses.

4.3 The execution process

The execution process uses the temporal dependency graph as an executable data structure that monitors the activation of operations. The execution process consists of three main modules, namely *initialization, triggering*, and *activation*, as follows.

Initialization: Each transaction contains a sequence of signal events and operations that are initiated by the user. Each update operation is decomposed into primitive operations $u_1, \ldots, u_n$, where each primitive operation is designated to update a single variable or object. A *transaction subgraph* is constructed, consisting of all the operations that should be activated as a result of the activation of the primitive operations. An operation o is activated as a result of an update operation u_i iff there exists a path in the temporal dependency graph from u_i to o. For example, Figure 5 is a transaction subgraph of a transaction that was initiated as a result of signaling a *Pass-Traffic-Course* event.

Triggering: An operation is triggered once its predecessors in the transaction subgraph have terminated. A triggered operation consults the transaction subgraph to assess its status. If there are no more incoming edges to be triggered, then the operation is activated; otherwise, the operation waits.

Activation: The activation of an operation entails the retrieval of values, the call for auxiliary programs, and the calculation of formulae to decide upon new values to be inserted to the database. Upon completion of these phases, the database is updated, and any outgoing edges are triggered.

For example, On January 29, 1995, a traffic school course has concluded, and a transaction is issued, updating the database, to confirm that ten drivers has passed. This is done by signaling a *Pass-Traffic-Course* event. The transaction is as follows.

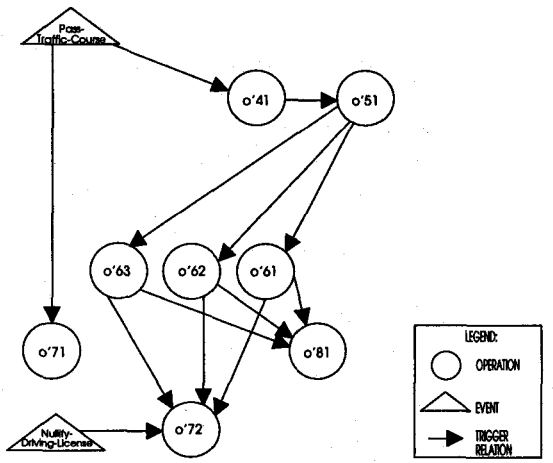

Fig. 5. A transaction subgraph

Begin transaction
Signal Pass-Traffic-Course for
Driver.Driving-License-Number=12345678,
...,
Driver.Driving-License-Number=34467978,
End transaction

The transaction is transformed into primitive operations. Figure 5 represents a transaction subgraph of the given transaction. For each driver, (o'_{41}) retrieves the valid value of *Violation-Points* of each traffic violation the driver committed. If it is not 0, then a new value of the *Violation-Points*, with the value 0 and validity [Jan 29 1995, ∞) is generated. The edge $\langle (o'_{41}), (o'_{51}) \rangle$ is triggered, and for each driver, (o'_{51}) derives *Points* to be 0 in the interval [Jan 29 1995, ∞). The edges $\langle (o'_{51}), (o'_{61}) \rangle$, $\langle (o'_{51}), (o'_{62}) \rangle$, $\langle (o'_{51}), (o'_{63}) \rangle$ are then triggered. Of the three operations, (o_{61}), (o_{62}), and (o_{63}), (o_{61}) is the only one that generates new values, since the precondition of processing (o'_{62}) and (o'_{63}), requiring the number of demerit points to be more than 6, is not valid. For each driver, (o'_{61}) derives *Traffic-Course-Deadline* to be "null" in the interval [Jan 29 1995, ∞). The edges $\langle (o'_{61}), (o'_{72}) \rangle$ and $\langle (o'_{61}), (o'_{81}) \rangle$ are then triggered. (o'_{71}) renews the driving-license of each of the drivers whose driving-license was revoked. As a result, a new value of the *Driving-License?* property, with the value "True" and a validity of [Jan 29 1995, ∞), is generated for each driver. (o'_{72}) does not result in any new value, since *Traffic-Course-Deadline* is nullified for each driver. (o'_{81}) is triggered with the names of the drivers and a nullified driving course deadline. At this point, there are no outgoing edges to trigger, and the transaction is terminated.

The execution process preserves the database consistency [10]. Therefore, applying a transaction on a database Δ that all of its operation clauses are satisfied, results in a database Δ' in which all operation clauses are satisfied as well. In addition, to handle the problem of long transactions [4], the update process minimizes the time required for processing a transaction, due to the use of a temporal dependency graph [10].

5 The properties of TALE

This section discusses three of TALE properties, namely active properties (Section 5.1), temporal properties (Section 5.2) and linguistic properties (Section 5.3).

5.1 Active capabilities of TALE

TALE is aimed at defining operations that contain valid times. However, TALE is also a rule language for active databases. As such, every ECA statement[5] has an equivalent operation clause in TALE. Consider the following ECA rule:

Event *ev*
Condition *cond*
Action *prog*

An equivalent operation clause in TALE is written as follows:

- If *prog* is an updating program that updates the elements $\{el_1, \ldots, el_n\}$:
  $\{el_1, \ldots, el_n\}$:= *prog triggered-by ev when cond.*
- If *prog* is not an updating program:
  apply prog triggered-by ev when cond.

While TALE subsumes ECA rules, the uniqueness of the model over other active database languages, is the combination of the capability of representing time dependent operations (Section 5.2), and reflective capabilities (Section 5.3) besides the standard reactive capabilities of active databases. Some other issues that are not discussed in this paper relate to composite events and coupling modes. The issue of composite events is an issue orthogonal to the temporal capabilities as discussed in this paper. Therefore, any model of composite events definition and manipulation (e.g. [3]) can be adopted by TALE. The default coupling mode used in TALE falls into the *deferred* category. An extension to support other coupling modes is not discussed in this paper. Conflict resolution among different operations are resolved by syntactic restriction as well as by imposing a totally ordered priority.

[5] The term *ECA statement* refers to a *simple ECA rule* [3].

5.2 Temporal capabilities of TALE

TALE enables three temporal effect types on operations.

Derived valid-time temporal projection: An activation of an operation results in the generation of values with bounded temporal validity. For example, consider the operation clause $x := y + 3$, and assume that x and y are properties of the same class. On January 1994, $\alpha.y$ is modified to be 7, valid in [Jan 1 1994, Jan 8 1994). As a result, $\alpha.x$ is modified to be 10, valid in [Jan 1 1994, Jan 8 1994). The valid time is derived as an intersection of the valid times of the participants' instances.

Temporal conditions: An operation consists of a condition expression that can take temporal variables as arguments. For example:

x:= y+3 *when* $\langle$z, $[t_s - 365, t_e)\rangle$ >100
2 * y+8 *otherwise*

$t_v = [t_s, t_e)$, is the valid time of the derived value. The condition is a function of the start time (t_s) and end time (t_e) of that interval. Assume that x, y and z are properties of the same class. On January 1 1994, $\alpha.y$ is modified to be 7, valid during [Jan 1 1994, Jan 8 1994). $\alpha.x$ is modified to be 10, valid during [Jan 1 1994, Jan 8 1994), only if the value of $\alpha.z$ was more than 100 throughout the interval [Jan 1 1993, Jan 8 1994). Otherwise, $\alpha.x$ is modified to be 22, valid during [Jan 1 1994, Jan 8 1994). These conditions are depicted in the temporal dependency graph as temporal elements associated with a triggering edge.

Explicit valid-time temporal projection: An operation can result in the generation of values with bounded temporal validity that is a function of the temporal validity of other values. For example, the operation clause $x := y + 3$ *valid-in* $[t_s, t_e + 4)$ defines an explicit valid time for the derivative. On January 1 1994, $\alpha.y$ is modified to be 7, valid in [Jan 1 1994, Jan 8 1994). As a result, $\alpha.x$ is modified to be 10, valid in [Jan 1 1994, Jan 12 1994). Explicit valid-time temporal projection is depicted in the temporal dependency graph as a temporal element associated with an update edge.

TALE presents new temporal capabilities. To emphasize these capabilities, other common temporal aspects are kept to the minimum. For example, a natural extension of this language, not discussed in this paper, is the support of temporal operators. Ongoing work focuses on an extensive temporal support in the TALE framework.

5.3 Reflective programming capabilities

TALE employs both the reactive programming style of active databases, and a higher level approach of reflective programming, according to which implicit properties of operations define their relationships with the database. As discussed in the previous section, the activation of an operation o depends on the

contents of the TRIGGER-SET(o). The contents of this set can be inferred by the operation definition and does not require an explicit definition. For example, consider a derive operation o that uses the operation clause $x := a + b$. The implicit contents of TRIGGER-SET(o) is $\{a, b\}$. Thus, any modification to an instance of either a or b activates o. By definition, when the TRIGGER-SET(o) is inferred, it contains all the participants, thus REQUEST-SET(o)=$\emptyset$.

The TRIGGER-SET(o) and the REQUEST-SET(o) sets are inferred, unless the TRIGGER-SET(o) is defined explicitly. For example, the operation clause $x := a + b$ *triggered-by* a, (which means the event "a is updated"), defines an explicit TRIGGER-SET(o). In these cases, the REQUEST-SET(o) is defined as the set of all participants that are not included in the TRIGGER-SET(o).

6 Conclusion

In this paper we have presented TALE, a Temporal Active Language and Execution model, that encapsulates a wide range of inter-relationships of temporal and active capabilities. TALE language primitives define operation clauses, a behavioral description of executable entities called operations. TALE handles operations that define the relationships among values and time-stamps of variables in a database, in addition to other types of operations. Operation clauses are defined in a declarative fashion, with the possible use of auxiliary programs. Temporal relationships in operation clauses are given through the use of participants, a valid time bound properties of the form $\langle$property-name, $t_v\rangle$. Each value in the temporal active database has a bounded temporal validity, that is either given explicitly by the user or derived through the use of operation clauses. TALE extends active databases (that traditionally support temporal events), to support also temporal conditions and temporal actions. Consequently, TALE provides an easy-to-use built-in temporal capabilities in an active database environment. In the degenerate case, when TALE works on a non-temporal active database, TALE would be able to represent ECA rules, as demonstrated in the previous section.

The use of TALE in complex applications facilitates the programming and debugging tasks, and assists in creating a reliable environment for applications that use retroactive and proactive updates. A prototype of a system that is based on TALE is currently under development on the basis of MS-Access 2.0 for Windows, in the Windows 3.11 environment. A temporal database model is mapped to the MS-Access environment using an extension to the Temporal Normal Form (TNF) [18], in which a relation consist of all the variables that share the same time stamp. An ongoing work focuses on the parallel execution model of the update process of a temporal active database, and the use of parallel execution to support schema versioning [12]. An additional work focuses on an extensive temporal support in the TALE framework.

References

1. A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence and observable determinism. In *Proceedings of ACM SIGMOD*, pages 59–68, June 1992.
2. S. Chakravarthy and D. Mishra. An expressive event specification language for active databases. *Data and Knowledge Engineering*, 13(3), Oct. 1994.
3. U.S. Chakravarthy. Rule management and evaluation: An active DBMS perspective. *ACM SIGMOD Record*, 18(3):20–28, Sep 1989.
4. U. Dayal, M. Hsu, and R. Ladin. A transaction model for long-running activities. In *Proceedings of the 17th VLDB*, pages 113–122, Sep 1991.
5. K.R. Dittrich and S. Gatziu. Time issues in active database systems. In R.T. Snodgrass, editor, *Proceedings of the International Workshop on Infrastructure for Temporal Databases*, Arlington, TX, June 1993.
6. O. Etzion. Active interdatabase dependencies. *Information Sciences*, 75:133–163, 1993.
7. O. Etzion. The reflective approach for data-driven rules. *International Journal of Intelligent and Cooperative Information Systems*, 2(4):399–424, December 1993.
8. O. Etzion and B. Dahav. Self-stabilization in databse consistency maintenance. Technical Report ISE-TR-95-1, Technion-Israel Institute of Technology, Feb 1995.
9. O. Etzion, A. Gal, and A. Segev. Temporal active databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Database*, June 1993.
10. A. Gal. *TALE — A Temporal Active Language and Execution Model.* PhD thesis, Technion—Israel Institute of Technology, Technion City, Haifa, Israel, May 1995. Available through the author's WWW home page, http://www.cs.toronto.edu/~avigal.
11. A. Gal and O. Etzion. Maintaining data driven rules in databases. *IEEE Computer*, 28(1):28–38, Jan 1995.
12. A. Gal and O. Etzion. A parallel execution model for updating temporal databases. to appear in the International Journal of Computer Systems Science and Engineering, 1995.
13. A. Gal, O. Etzion, and A. Segev. A language for the support of constraints in temporal active databases. In *Proc. Workshop on Constraints, Databases and Logic Programming*, pages 42–58, Portland, Oregon, Dec 1995.
14. N. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases. In *International Conference on Very Large Databases*, Vancouver, Canada, Aug 1992.
15. S. Hudson and R. King. CACTIS: A database system for specification functionality defined data. In *Proceedings of the IEEE OOBDS Workshop*, pages 26–37, Sep 1986.
16. C.S. Jensen et al. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–63, 1994.
17. M.R. Klopprogge and P.C. Lockmann. Modeling information preserving databases; consequences of the concept of time. In *Proceedings of the International Conference of VLDB*, Florance, Italy, 1983.
18. S.B. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49:147–175, 1989.
19. N. Pissinou. Towards an infrastructure for temporal databases—A workshop report. *ACM SIGMOD Record*, 23(1):35, 1994.

20. N.L. Sarda. HSQL: Historical query language. In *Temporal Databases*, chapter 5, pages 110–140. The Benjamin/Commings Publishing Company, Inc., Redwood City, CA., 1993.
21. R. Snodgrass et al. TSQL2 language specification. *ACM SIGMOD Record*, 23(1):65–86, Mar 1994.
22. S.Y.W. Su and H.M. Chen. A temporal knowledge representation model OSAM*/T and its query language OQL/T. In *Proceedings of the International Conference on VLDB*, 1991.
23. G.T.J. Wuu. SERQL: An ER query language supporting temporal data retrieval. In *The Proceedings of the 10th International Pheonix Conference on Computers and Communications*, Mar 1991.

A Appendix: The EBNF form of the Language's Grammar

The EBNF form of the Language's Grammar is as follows.

Operation-Clause	::= *derive* Derivation-Operation
	| *enforce* Enforce-Operation
	stabilizer= Derivation-Operation
	| *apply* Apply-Operation
Derivation-Operation	::= Derived-Variable := Derivation-Formula
	| Set-Derived-Variable := Set-Derivation-Formula
Derivation-Formula	::= Derivation-Expression
	| Derivation-Clause {; Derivation-Clause }*
	[; Derivation-Expression *otherwise*]
Set-Derivation-Formula	::= Set-Derivation-Expression
	| Set-Derivation-Clause { Set-Derivation-Clause }*
	[Set-Derivation-Expression *otherwise*]
Enforce-Operation	::= Enforce-Expression
	| Enforce-Clause { Enforce-Clause }*
	[Enforce-Expression *otherwise*]
Apply-Operation	::= Apply-Expression
	| Apply-Clause { Apply-Clause }*
	[Apply-Expression *otherwise*]
Derivation-Clause	::= Derivation-Expression *when* Prerequisite
Set-Derivation-Clause	::= Set-Derivation-Expression *when* Prerequisite
Enforce-Clause	::= Enforce-Expression *when* Prerequisite
Apply-Clause	::= Apply-Expression *when* Prerequisite
Prerequisite	::= Cond-Spec
Derivation-Expression	::= Numeric-Exp Derive-Exp-Prefix
	| Routine-Call-Exp Derive-Exp-Prefix
	| Cond-Spec Derive-Exp-Prefix
Set-Derivation-Expression	::= Numeric-Exp Exp-prefix
	| Routine-Call-Exp Exp-prefix
	| Cond-Spec Exp-prefix
Enforce-Expression	::= Cond-Spec Exp-Prefix
Apply-Expression	::= Routine-call-Exp Exp-Prefix
Derive-Exp-Prefix	::= Exp-Prefix [*valid-in* Temporal-Element]
Exp-Prefix	::= [*triggered-by* Event-list] [*signals* Event-list]
Routine-call-Exp	::= Routine-Name *(* {Param}* *)*
Numeric-Exp	::= Data-Element
	| Numeric-Exp Numeric-Oper Numeric-Exp
	| *(* Numeric-Exp *)*
Derived-Variable	::= Property
Set-Derived-Variable	::= {participant}*
Data-Element	::= Property
	| Participant
	| Sequence-Oper
	| Constant
Sequence-Oper	::= Seq-Func*(* Seq-Spec *)*
Seq-Spec	::= Seq-Def

	| Mod-Seq
Mod-Seq	::= Seq-Def Numeric-Oper Numeric-Exp
Cond-Spec	::= [Sign] Cond-Unit { Connective [Sign] Cond-Unit }*
Cond-Unit	::= Data-Element Binary-Pred Numeric-Exp | Seq-Spec
Participant	::= Data-Element Temporal-Element
Property	::= Identifier
Routine-Name	::= Identifier
Seq-Def	::= Identifier
Binary-Pred	::= = | < | > | ≤ | ≥ | ≠
Seq-Func	::= *sum* | *max* | *min* | *first* | *last* | *count*
Numeric-Oper	::= + | − | * | / | **
Sign	::= ¬
Connective	::= ∧ | ∨
Constant	::= {x | x is a number } | {x | x is a string } | *now*
Param	::= Identifier
Event-list	::= Event {, Event}*
Event	::= Property | Identifier
Temporal-Element	::= Time-Point | Time-Interval | { Temporal-Elements }*
Time-Interval	::= [Time-Point, Time-Point)
Identifier	::= LABEL
Time-Point	::= {t | t is a time-stamp }