

A Multiagent Update Process in a Database with Temporal Data Dependencies and Schema Versioning

Avigdor Gal and Opher Etzion, *Member, IEEE Computer Society*

Abstract—Temporal data dependencies are high-level linguistic constructs that define relationships among values of data-elements in temporal databases. These constructs enable the support of schema versioning as well as the definition of consistency requirements for a single time-point and among values in different time-points. In this paper, we present a multiagent update process in a database with temporal data dependencies and schema versioning. The update process supports the evolution of dependencies over time and the use of temporal operators within temporal data dependencies. The temporal dependency language is presented, along with the temporal dependency graph—which serves as the executable data structure. A thorough discussion of the feasibility, performance, and consistency of the presented model is provided.

Index Terms—Temporal databases, data dependencies, database updates, parallel processing, schema versioning, high-level languages, active databases.



1 INTRODUCTION AND MOTIVATION

THE phenomenon of *data dependencies* in databases occurs when a value of a data-element is dependent upon values of other data-elements. A data dependency—an expression also known in the literature [42] as *value dependency*—can be materialized by means of a **derivation**, i.e., a data element whose value is a calculated function of other data-elements' values:

$$\text{Program.Advertisement - Price} := \text{Price} - \text{Formula}(\text{Program.Hour}, \text{Program.Rating}) \quad (1)$$

For example, Expression 1 is a derivation. Any modification of the instances referred to in the derivation's right hand side triggers the recalculation of relevant instances of the data-element that resides in the left hand side. A data-element whose value's modification triggers the recalculation is called *dependee* (e.g., *Program.Rating*), and the data-element whose value is updated is called *dependent* (e.g., *Program.Advertisement-Price*).

Throughout the history of database research there has been a considerable interest in the modeling and efficient implementation of data dependencies. The DBTG proposal [9] requires the support of derived data as a virtual term. The *view* concept as a virtual term was part of the early relational database model. More recent studies [4], [18] argue that materialization of derived data is desirable in some cases for the sake of efficiency (e.g., frequent retrieval requests for the derived data vs. infrequent updates) as well as effectiveness reasons (e.g., derived values that should be

utilized in real-time). Various models use the concepts of *integrity constraints* [8] and *derived data* [34] to materialize data dependencies. The feasibility of derivation materialization depends upon the ability to efficiently compute materialized data dependencies, as discussed in [29], [41], [5].

Recent trends in database programming deal with the explicit specification of data dependencies and their separation from application programs. As claimed in [8], "It leads... to much more compact and understandable application programs." Explicit specification of data dependencies started with the *semantic data models* [28] and carried on to other areas such as active databases [7] in which event driven behavior is being made explicit.

One of the main results of working with data-dependencies programming is the observation that there are strong relationships between data dependencies and the concept of time. For example, the combination between dependencies and time is vital for decision support and decision analysis systems that maintain information about the past (for analysis) and information about the future (for planning) with possibly different schema versions. Prediction about future values of a data-element may be dependent upon past and present values of data-elements. New information can be obtained about past and future values, resulting in the retroactive and proactive updates of the database. For example:

The advertising prices in commercial television are retroactively determined as a function of programs' actual rating average over a two months period. The price formula pf_1 that had been valid for many years was changed to the price formula pf_2 in January 1995. In February 1995 a retroactive update was issued for the rating figures of December 1994. This update should result in an adjustment to the advertising price for December 1994, according to pf_1 , and in an adjustment to the advertising price for January 1995 according to pf_2 . The required functionality is concurrent

- A. Gal is with the Department of MSIS, Rutgers University, 94 Rockefeller Rd., Piscataway, NJ 08854-80. E-mail: avigal@rci.rutgers.edu.
- O. Etzion is with the IBM-Haifa Research Laboratory, Matam 31905, Haifa, Israel. E-mail: opher@haifa.vnet.ibm.com.

Manuscript received 3 Oct. 1994; revised 20 Feb. 1996.

For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104425.

treatment of two data-dependency versions and the inclusion of time concepts in data-dependency definitions.

These capabilities were identified as essential in the process of transfer from *data engineering* toward an *information engineering* that is aimed at employing the information resource for corporations' policy making [48]. In the current technology, these functionalities are difficult to achieve and require ad hoc application programs with no abstractions to support them.

In this paper, we present an execution model for a database that supports temporal data dependencies. The combination of the functionality of data dependencies and the time dimension is realized by satisfying two major requirements, **the evolution of dependencies over time** and **the use of temporal operators within the definition of data dependencies**.

1.1 Evolution of Dependencies Over Time

Dependencies can evolve in a similar fashion to schema evolution and the evolution of application programs. Each dependency may have several versions that are applicable in different time points. For example, the **advertisement price** is calculated according to one formula in 1994 and another formula in 1995. The evolution of dependencies, coupled with the possibility to update past and future values, requires the capability of identifying the relevant data dependency that is applicable for each update operation. Furthermore, when dealing with update operations that may span over an interval, different portions of this interval may conform with diverse data dependencies. The support of evolving dependencies enables the modification of past or future values while maintaining a consistent view of database versions in a dynamic environment. This support is valuable for any system in which knowledge about past or future events should be recorded and analyzed.

1.2 Use of Temporal Operators within the Definition of Data Dependencies

The definition of data dependencies can be extended to materialize the temporal effect in the following two ways:

- 1) The dependent data-element and the dependee data-element may have different valid times. For example, a salary is updated at the end of the year according to values of bonuses that are calculated at the end of each quarter. A modification of a bonus value updates the future value of a salary and not the current one.
- 2) The update of a derived value, performed at a given time point t , entails the retrieval of values whose valid time is different than t . For example, when an employee is promoted from a junior manager to a senior manager, the operation that calculates the initial salary retrieves the number of employees that reported to this manager at all times. This salary is being calculated only when the promotion occurs, but it uses values that have been valid throughout the history.

The execution model presented in this paper, is based upon the following three components:

- 1) The temporal data dependencies are defined and compiled into a *temporal dependency graph* that is an extension of the data-dependency graph developed in the PARDES model [17], [18]. The execution process, based on the temporal dependency graph guarantees the consistency of the resulting database in a minimal number of update operations.
- 2) A parallel processing execution [11] using temporal agents and a multiagent execution model is provided when a single update operation refers to a time interval that spans over the valid time of more than one schema version, or a collection of update operations that reside in the same transactions refer to distinct schema versions. Temporal agents are *information agents* [37] that operate within different time frames, and perform predefined and inferred actions that interpret the update operations and their consequences according to the valid schema in a given time point.
- 3) Each transaction may be partitioned among parallel processes, such that each process executes a temporal agent. Conflicts are avoided by a communication protocol among the temporal agents.

In this paper, we discuss the multiple schema versions implementation and the interconnections protocol among temporal agents. Section 2 describes the temporal database structure and the dependency language. The update process and the concepts of temporal agents and a temporal dependency graph are given in Section 3. The model's properties are discussed in Section 4, and a comparison of this study with related work is given Section 5.

2 THE SYSTEM DESIGN PRIMITIVES

This section presents the building blocks used by the system designer to construct the metadata that contains the temporal database model and the temporal data dependencies. The temporal data model is presented in Section 2.1. The temporal data-dependency language is presented in Section 2.2. Section 2.3 presents the relationships between the database schema and the operations that are inferred from the defined temporal data dependencies. These relationships are used for runtime execution monitoring and as a verification tool for the system designer.

2.1 The Temporal Data Model

The data model is a bitemporal data model [32] supporting valid time and transaction time. *valid time* (t_v) designates the time points at which the value is considered to be true in the modeled reality, expressed using a *temporal element* [20], which is a finite union of time intervals (e.g., [Jan 1995, June 1995] \cup [Aug 1995, Feb 1996]).¹ *Transaction time* (t_x) designates the time when a value becomes current in the database. This time type may be implemented by using a transaction commit time.

The data model employed in this paper can augment any data model that supports operations at the attribute

1. It is common in temporal databases to use discrete time representation, based on discrete atomic unit of time called chronon [31]. Consequently, an interval that includes a single chronon t can be represented as the interval $[t, t + 1)$.

Class =	Distributor	Class =	Distributor Rank
Properties =	<u>Distributor-Number</u>	Properties =	<u>Distributor-Rank-Code</u>
	Name		Low-Commission-Rate
	Distributor-Rank		Medium-Commission-Rate
	Salary		High-Commission-Rate
	Number-of-Subscribers [derived]		Medium-Lower-Bound
	Distributor-Status [derived]		High-Lower-Bound
	Commission [derived]		Upper-Bound
	Annual-Commission [derived]		Subscribers-Limit
			Total-Dividend
Class =	Subscriber	Class =	Metadata-Changes
Properties =	<u>Subscriber-Number</u>	Properties =	<u>Change-Code-Number</u>
	Name		Property-Name
	Address		Property-Structure
	Zipcode		
	Assigned-Distributor [derived]		
	Expiration-Date		

Fig. 1. Schema definition.

level. In particular, one may assume an object-based model, in which each class has properties, or a relational model, in which each relation has columns. The term *class* defines either a class in the object-based model, or a relation in the relational model. The term *object* defines an instance of a class or a tuple in a relation. *Property* defines an attribute in the object-based model or a column in the relational model. The flexibility in choosing the data model results in a *data-model independence*, in which the temporal effect can be added on top of existing systems.

A class definition contains the specification of properties that are applicable to its instances, along with their types. For example, Fig. 1 presents a partial schema definition of the distributors' case-study. It consists of four classes:

- 1) the Distributor,
- 2) the Subscriber,
- 3) the Distributor Rank, and
- 4) the Metadata Changes.

The Distributor and the Subscriber classes represent personal details of distributors and subscribers, respectively. The Distributor-Rank class contains details that are applicable to each distributor possessing that rank. The Metadata-Changes class is a metadata class that stores changes made in the schema level. [derived] denotes a derived property, i.e., a property whose value is derived using a temporal data dependency. Further discussion of derived properties is presented in Section 2.2.

A *variable* is an instance of a property in the sense that it is associated with a specific object. At each time point, a variable is composed of one or more values, each of which is valid in a given time interval. Although each variable may have temporal characteristics, the system designer may choose to eliminate the temporal characteristics for some of the variables due to storage limitations or irrelevance considerations.

A variable in a temporal database consists of a set $\{se_1, \dots, se_n\}$ of n state-elements. A *state-element* represents the variable's value along with the value's time stamp. A

state-element is a tuple of the form $\langle se-id, value, t_x, t_v \rangle$,² where *se-id* is a unique identifier of the state-element, *value* represents the variable's value (an atom, a set, a sequence, a tuple or a reference to another object), and t_x and t_v are the transaction time and valid time, respectively.

An object is represented as a set of variables. An object's *state* is a set of all its variable states. An object is identified by a subset of the object's state, referred to as the *object identifier* (e.g., a primary key in the relational model). For example, the underlined properties in Fig. 1, are the object-identifiers.³

Fig. 2 illustrates the changes to the state of the variable Zipcode of the Subscriber John Doe. The term ∞ in a t_v of a state-element means that the state-element is assumed to be valid forever unless other information is provided. Other state-elements can be added with intersecting valid time intervals. If several state-elements have intersecting valid time intervals, the state-element with the highest t_x is considered as the valid one.⁴

Zipcode =				
(s1)	12345,	Dec 1 1990,	[Dec 1 1990, ∞)	
(s2)	12445,	Nov 12 1991,	[Jan 1 1992, ∞)	
(s3)	1w2zk,	<u>June 5 1993,</u>	<u>[Sep 1 1993, ∞)</u>	
		t_x	t_v	

Fig. 2. State-elements of John Doe's Zipcode.

2.2 Temporal Data-Dependency Language

In this section, we present the temporal data-dependency language (the term *dependency* is used in this paper interchangeably with the full term). We start by presenting the PARDES language in Section 2.2.1, and proceed with

2. Additional attributes of information about source, validity, accessibility, etc., can be added to a state-element. These extensions are discussed in [24].

3. Using object-based implementation environment, each object is also associated with a unique object identity.

4. This is a common assumption based on the fact that the last update overrides previous updates [19]. Different assumptions are also possible.

the language extension to temporal data dependencies in Section 2.2.2. An example of dependencies in the case study is presented in Section 2.2.3.

2.2.1 PARDES Language

The temporal data-dependency language is an extension of the PARDES language [17], a declarative language for data dependencies. A data-dependency expression in PARDES is a declarative definition that uses arithmetic operators (e.g., $y := 3 * x + z$), or functions (e.g., $y := f(x, w, z)$), or binary predicates (e.g., $y > x + w$).

The language's semantics eliminate conflict resolution cases by allowing each property to appear at the left hand side of a single data dependency. Thus, all the different cases of deriving values of a property are encapsulated within a single data dependency. Each data dependency may contain conditions that are assumed to be totally ordered. For example, in the data dependency $y := 3 * x + z$ **when** C1 ; $2 * x - z$ **when** C2, the first condition C1 is evaluated first and only if it is not satisfied, the second condition C2 is evaluated. Consequently, the second condition is interpreted as C2 and not C1. These restrictions guarantee unique and deterministic execution for each case. The conditions supported by the language are binary predicates ($=, \leq, \geq, \neq, <, >$) and set membership operations.

Data dependencies are preserved either in a data-driven or event-driven fashion. The default case is data-driven, in which preserving operations are activated when any instance of a dependee is modified. In this case, the data dependency can be viewed as an *invariant* that should be preserved at all times. Alternatively, data-dependency preservation operations can be activated as a response to an event detection. 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, or a sensor's output, or a change in a variable. An event can be either atomic or composite. A thorough discussion of events' classification and types can be found in [6], [12]. Event driven data dependencies are specified using an additional clause **triggered-by**, that designates the activating event. For example, in the data dependency $y := 3 * x + z$ **triggered-by** *ev*, the dependency preservation operation is activated only when the event *ev* is detected. The language supports simple events and several types of complex events (conjunction, disjunction and sequencing). An extension of the event language to support more types of complex events (such as the types mentioned in [6]) is under construction.

2.2.2 The Temporal Extensions to PARDES

The PARDES dependency language is extended to support the temporal extension in the following two forms:

- 1) The dependent data-element and each dependee may have valid time restrictions. For example, consider the following dependency:

$$x := \langle a, t_{v2} \rangle + \langle b, t_{v3} \rangle \text{ valid-in } t_{v1} \text{ when } C1 \\ \langle c, t_{v2} \rangle + \langle d, t_{v3} \rangle \text{ valid-in } t_{v4} \text{ otherwise}$$

The value of x in t_{v1} is calculated as the sum of the values of a in t_{v2} and b in t_{v3} , when $C1$ is evaluated to

be true. Otherwise, the value of x in t_{v4} is calculated as the sum of the values of c in t_{v2} and d in t_{v3} .

- 2) A variable is updated using the dependency that is valid for that update. For example, if there are two versions of the dependency d_i , such that version k is valid during 1994, and version l is valid during 1995, a retroactive update issued in 1995 which updates data during 1994, should use version k of d_i .

2.2.3 Temporal Data Dependencies—An Example

Fig. 3 presents a partial example of temporal data dependencies in the case study domain. The first four dependencies denote integrity constraints. (d1) restricts the number of subscribers per distributor. (d2) and (d3) define the relationships among the low, medium and high commission rates. (d4) prevents an increase of more than 10 percent in a distributor's salary, during a successive six months' period.

The dependencies (d5)-(d9) are derivations. Any change in values of a variable may trigger an operation. (d5) computes the number of subscribers as an aggregate function over the Subscriber class. (d6) determines the distributor status as a function of the number of subscribers assigned to a distributor. (d7) calculates the distributor's commission. The calculation is based on the distributor's status, the number of subscribers assigned to the distributor, and the commission rates. (d8) calculates the annual commission based on the performance of the distributor in the past year. The low annual commission is given in two months delay and the medium annual commission is given in one month delay. (d9) uses a function for reallocating subscribers to distributors. It updates the Assigned-Distributor property and it is triggered by changes in the Zipcode and Expiration-Date properties. The retrieved properties are given in the parenthesis, and the updated property is the dependent value that appears in the left-hand side of the dependency definition.

When the participating properties are not of the same class, a *matching* process is required. The matching process uses additional information to determine the participating instances. For example, in (d1) the matching can be based on a comparison between the values of Distributor-Rank in the Distributor class and Distributor-Rank-Code in the Distributor-Rank class. A thorough discussion of the matching process can be found in [16].

Each temporal data dependency is translated by an inference mechanism to an update program that deals with all the dependency's implications. The inferred program is referred to as a *dependency preserving operation* (we use the short term *operation* interchangeably with *dependency preserving operation*). The symbols (o1), (o2), ..., (o9) denote the operations that have been inferred from the dependencies (d1), (d2), ..., (d9).

2.3 The Relationships Between Operations and Schema Elements

Relationships between operations and schema elements are inferred by the translation process. The relationships of an operation o with the schema elements are defined through the use of the following three sets:

(d1) Number-of-Subscribers	\leq	Subscribers-Limit
(d2) Low-Commission-Rate	\leq	Medium-Commission-Rate
(d3) Medium-Commission-Rate	\leq	High-Commission-Rate
(d4) $\langle \text{Salary}, [t, t + 6\text{month}] \rangle$	\leq	$1.1 * \langle \text{Salary}, [t, t + 1] \rangle$
(d5) Number-of-Subscribers	$:=$	count (Subscriber)
(d6) Distributor-Status	$:=$	'Low' when Number-of-Subscribers < Medium-Lower-Bound 'Medium' when Number-of-Subscribers < High-Lower-Bound 'High' otherwise
(d7) Commission	$:=$	Number-of-Subscribers * Low-Commission-Rate when Distributor-Status = 'Low' Number-of-Subscribers * Medium-Commission-Rate when Distributor-Status = 'Medium' Number-of-Subscribers * High-Commission-Rate otherwise
(d8) Annual-Commission	$:=$	0.01 * Total-Dividend valid-in [NOW + 2month, NOW + 1year) when $\langle \text{Distributor-Status}, [\text{NOW} - 1\text{year}, \text{NOW}] \rangle = \text{'Low'}$ triggered-by Begin-of-Year 0.02 * Total-Dividend valid-in [NOW + 1month, NOW + 1year) when $\langle \text{Distributor-Status}, [\text{NOW} - 1\text{year}, \text{NOW}] \rangle = \text{'Medium'}$ triggered-by Begin-of-Year 0.05 * Total-Dividend valid-in [NOW, NOW + 1year) when $\langle \text{Distributor-Status}, [\text{NOW} - 1\text{year}, \text{NOW}] \rangle = \text{'High'}$ triggered-by Begin-of-Year
(d9) Assigned-Distributor	$:=$	Heuristic-Assignment (Zipcode, Expiration-Date, Number-of-Subscribers) triggered-by {Zipcode, Expiration-Date}

Fig 3. Dependencies in the case study.

- 1) UPDATE-SET(o) is defined as the set of pairs $\langle p, t_v \rangle$, such that p is a property that can be updated as a result of the activation of o and t_v is a temporal element in which p is updated. UPDATE-SET(o) is inferred as the dependent property, when an operation o derives new state-elements.
- 2) TRIGGER-SET(o) is defined as the set of pairs $\langle p, t_v \rangle$, such that p is a property or an event that can activate o , and t_v restricts the temporal range in which o is activated. A change to p in t_v (if p is a property), or the detection of p (if p is an event) triggers o .
- 3) REQUEST-SET(o) is the set of pairs $\langle p, t_v \rangle$, such that p is a property that might get retrieved as a result of the activation of o . p cannot trigger an operation. Thus, $\text{TRIGGER-SET}(o) \cap \text{REQUEST-SET}(o) = \emptyset$.

$$x := \langle a, t_{v2} \rangle + \langle b, t_{v3} \rangle \text{ valid-in } t_{v1} \quad (2)$$

For example, let o be an operation that preserves the temporal data dependency of Expression 2, and assume that a , b , and x are properties of the same class. The relationship sets of o are:

- 1) $\text{UPDATE-SET}(o) = \{\langle x, t_{v1} \rangle\}$.
- 2) $\text{TRIGGER-SET}(o) = \{\langle a, t_{v2} \rangle, \langle b, t_{v3} \rangle\}$.
- 3) $\text{REQUEST-SET}(o) = \emptyset$

If $\alpha.a$ is modified in t_{v2} , then $\alpha.b$ in t_{v3} is retrieved, and $\alpha.x$ in t_{v1} is derived and updated.⁵ The activation of an operation o depends upon the contents of the TRIGGER-SET(o). The contents of this set are inferred using the dependency definition, and do not require an explicit definition. When the TRIGGER-SET(o) is inferred, it contains all the participants, thus $\text{REQUEST-SET}(o) = \emptyset$. The TRIGGER-SET(o) and the REQUEST-SET(o) are inferred, unless the TRIGGER-SET(o) is explicitly defined.

For example, the dependency of Expression 3 has an explicit TRIGGER-SET(o) = $\{\langle a, t_{v2} \rangle\}$. In this case, the REQUEST-SET(o) is defined as the set of all participants that are not included in the TRIGGER-SET(o), i.e., $\text{REQUEST-SET}(o) = \{\langle b, t_{v3} \rangle\}$.

$$x := \langle a, t_{v2} \rangle + \langle b, t_{v3} \rangle \text{ valid-in } t_{v1} \text{ triggered-by } \langle a, t_{v2} \rangle \quad (3)$$

The role of these three relationships is twofold. They are used to monitor the runtime execution, as discussed in Section 3, and they function as a validation tool, enabling the system designer to capture the global application behavior that is inferred from the various dependencies using a visualization tool.

⁵ Incremental calculations are possible in some cases [34]. Yet, this discussion is beyond the scope of this paper.

3 THE UPDATE MODEL

This section presents the update process of a temporal database that supports temporal data dependencies and schema versioning. The **evolution of dependencies over time** requirement is achieved using the concept of temporal agents, defined and discussed in Section 3.1. The dependencies and their relationships with the database are compiled into a *temporal dependency graph*, an executable data structure that monitors the execution of update operations. This concept is defined and discussed in Section 3.2. Section 3.3 presents the runtime execution process.

3.1 Temporal Agents

In this section, we present the concept of temporal agents as a means to support schema and dependencies versioning in temporal databases. Let DB be a database with an active time domain ATD . The active time domain consists of the collection of time-points for which there exists a relevant knowledge in the database. ATD is initialized to be the valid time of the first schema of the application and can be expanded later. $DB = D \cup MD$ consists of the data (D) and the metadata (MD) of the application.

The metadata of an application $MD = IS \cup IC$ consists of the initial schema (IS) and the incremental schema changes ($IC = ic_1, \dots, ic_n$) that were added to the initial schema. The database schema contains dependencies as well as other meta data entities. In the example presented in Fig. 1, IC is represented using the Metadata-Changes class.

Let s_i be a schema version. If $i = 0$ then $s_i = IS$. Otherwise, s_i is the schema version that is generated by applying the changes ic_1, \dots, ic_i to IS .

Each s_i has an associated time region, τ_i , the valid time temporal element in which s_i is applicable. The time region of a schema version s_i is the part of the t_v of ic_i that was not overridden by later changes. For each ic_i $t_v(ic_i)$ is explicitly specified in the update, for convenience reasons we define $t_v(ic_0) = t_v(IS)$. Given n schema changes ic_1, \dots, ic_n , τ_i ($0 \leq i \leq n$) is calculated as:

$$\tau_i = t_v(ic_i) - \bigcup_{j=i+1}^n t_v(ic_j).$$

Note that in some cases the second component of the above expression is empty, such as the cases in which the lower bound of the union is bigger than the upper bound. For example: $\tau_0 = t_v(IS)$ and $\tau_n = t_v(ic_n)$.

Let TAG_i be a temporal agent. TAG_i is an application program that preserves the database consistency with respect to s_i by executing the dependency preservation operations that are applicable in its time region (τ_i). TAG_i uses s_i to deduce the operations that should be activated to maintain the database consistency as a result of a user update or an event.

The number of temporal agents is identical to the number of the different versions of the schema that are relevant to the application. In practice [13], changes are accumulated and the generation of versions is not frequent. Thus, the number of versions is relatively small.

Each temporal agent TAG_i has a set CO_i of connectors. A *connector* in CO_i is a communication element of a temporal

agent TAG_i that allows message passing to other agents. The connectors are generated during the new schema introduction process. A connector is generated in the two following cases:

- 1) A variable v valid in schema version k that possibly triggers an operation o executed in the context of schema version l generates a sending connector that belongs to the agent TAG_k and a receiving connector that belongs to the agent TAG_l .
- 2) An operation o in a temporal dependency graph that belongs to a schema version l whose valid time is t_v and possibly updates a variable v valid in $t'_v \neq t_v$ generates a sending connector that belongs to the agent TAG_l and receiving connectors of all the schema versions that are valid during t'_v .

The temporal agents partition the time-regions each time the schema changes. Let ic_n be the n th incremental change of the initial schema. $t_v(ic_n)$ is the change's valid time. Fig. 4 presents the introduction of a new schema version algorithm.

For example, an active time domain of $[Dec\ 1\ 1990, \infty)$ is the initial time region of a temporal agent TAG_0 . A change ic_1 in the domain of Distributor-Rank with $t_v = [Jan\ 1\ 1992, \infty)$ is introduced. A new schema s_1 is generated along with a new temporal agent TAG_1 and a time region of $[Jan\ 1\ 1992, \infty)$. The time region of TAG_0 is now reduced to be the time interval $[Dec\ 1\ 1990, Jan\ 1\ 1992)$. An example that involves the generation of connectors is presented graphically later in this section.

Let n be the number of schema versions, $|oper_1|, \dots, |oper_n|$ be the number of operations in each of the schema versions, and $|prop_1|, \dots, |prop_n|$ be the number of properties in each of the schema versions. Each property is updated by a single operation. Hence, for each schema s_i , $|prop_i| \leq |oper_i|$. Consequently, the maximum number of properties and operations in a single schema version is bounded by $O(\max_{i=1}^n (|oper_i|))$.

According to Fig. 4, a pair of connectors, sco and $rco \in dest(sco)$ can be generated for each property in a relationship-set of an operation. Thus, the number of connectors in n schema versions is bounded by

$$2 \cdot \sum_{i=1}^n |oper_i| \cdot \sum_{i=1}^n |prop_i| = O((n \cdot \max_{i=1}^n (|oper_i|))^2).$$

Hence, the maximum number of metadata elements⁶ in a single schema version is bounded by $O((n \cdot \max_{i=1}^n (|oper_i|))^2)$.

Let s_n be a new schema version, generated by applying ic_n to s_{n-1} . The worst case time complexity of the introduction of a new schema version is bounded by

$$O(n \cdot (n \cdot \max_{i=1}^n (|oper_i|))^2).$$

We assume that the number of versions is smaller than the number of properties and operations, i.e., $\forall i : n < |oper_i|$. Therefore, even if $n \sim \max_{i=1}^n (|oper_i|)$, resulting in a time complexity of $O(n^4)$, where n is the number of schema

6. Metadata elements are schema elements, operations, and connectors.

Introduction of a new schema version algorithm

- 1) Add the n th incremental change of the initial schema to IC . $IC := append(IC, ic_n)$.
- 2) Generate a new schema s_n by applying ic_n to s_{n-1} .
- 3) Recompute $ATD := old(ATD) \cup t_v(ic_n)$, where $old(ATD)$ is the ATD that was valid before introducing ic_n .
- 4) Add a new temporal agent TAG_n to the application with $\tau_n = t_v(ic_n)$.
- 5) For each temporal agent TAG_i ($0 \leq i < n$), determine τ_i to be $\tau_i := old(\tau_i) - \tau_n$, where $old(\tau_i)$ represents the valid time region before the introduction of ic_n .
- 6) For each temporal agent TAG_i , such that $\tau_i \neq old(\tau_i)$ and for TAG_n , generate a set of connectors, as follows.

Let p be a property-name.
 Let t_v be a temporal element.
 Let o_i be an operation in s_i .
 Let o_n be an operation in s_n .

 - a) If $\langle p, t_v \rangle \in TRIGGER-SET(o_n) \vee \langle p, t_v \rangle \in REQUEST-SET(o_n)$, and $t_v \cap \tau_n \neq t_v$ then:

Generate a sending connector sco for each TAG_i such that $t_v \cap \tau_i \neq \emptyset$. Each sending connector in SCO_i has a valid time $t_v - \tau_i$ and a source = p .

Generate a receiving connector $rco \in dest(sco)$ for each sco , for TAG_n with valid time $t_v - \tau_n$ and a destination = o_n .
 - b) If $\langle p, t_v \rangle \in UPDATE-SET(o_i)$, and $t_v \cap \tau_i \neq t_v$, and $t_v \cap \tau_n \neq \emptyset$, then:

Generate a sending connector sco for TAG_i with a valid time $t_v - \tau_i$ and a source = o_i .

Generate a receiving connector $rco \in dest(sco)$ for TAG_n with valid time $t_v - \tau_n$ and a destination = p .
 - c) If $\langle p, t_v \rangle \in TRIGGER-SET(o_i) \vee \langle p, t_v \rangle \in REQUEST-SET(o_i)$, and $t_v \cap \tau_i \neq t_v$, and $t_v \cap \tau_n \neq \emptyset$, then:

Generate a sending connector sco for TAG_i with a valid time $t_v - \tau_i$ and a source = p .

Generate a receiving connector $rco \in dest(sco)$ for TAG_i with a valid time $t_v - \tau_i$ and a destination = o_i .
 - d) If $\langle p, t_v \rangle \in UPDATE-SET(o_n)$, and $t_v \cap \tau_n \neq t_v$ then:

Generate a sending connector sco for TAG_n with a valid time $t_v - \tau_n$ and a source = o_n .

Generate a receiving connector $rco \in dest(sco)$ for each TAG_i such that $t_v \cap \tau_i \neq \emptyset$. Each receiving connector rco has a valid time $t_v - \tau_i$ and a destination = p .
- 7) For each temporal agent TAG_i , such that $\tau_i \neq old(\tau_i)$, delete all the connectors with t'_v such that $t'_v \in old(\tau_i) - \tau_i$. For each connector in CO_i with t'_v such that $t'_v - (old(\tau_i) - \tau_i) \neq \emptyset$ assign new valid time, $t'_v = old(t'_v) - (old(\tau_i) - \tau_i)$

Fig. 4. Introduction of a new schema version.

versions, the introduction of a new schema version is still computationally feasible.

An important property of the temporal agent model is that the temporal agent's operations issue an equivalence class with respect to time points in the active time domain. This property is a consequence of Proposition 1.

PROPOSITION 1 (Partition of the active time domain). *The time regions of all the temporal agents, present at a given time t , are a partition of the active time domain.*

The complete proof of Proposition 1 is straightforward by using induction on the number of schema versions, and can be found in [21]. It requires the proof of the following two assertions, related to the collection of times in which a schema version s_i is valid (τ_i).

$$\text{Ass1: } \bigcup_{i=0}^n \tau_i = ATD$$

$$\text{Ass2: } \forall i, j (0 \leq i < j \leq n) \tau_i \cap \tau_j = \emptyset$$

3.2 The Temporal Dependency Graph

The *temporal dependency graph* is an executable data structure employed to preserve the consistency of a database with respect to its temporal data dependencies. When a transaction is issued, either explicitly or by an event detection, some

of the database's operations should be activated. These operations are activated according to the nodes traversed in the temporal dependency graph. The nodes of a temporal dependency graph designate events, operations and connectors in the application domain. As explained in Section 3.1, a connector is a communication element of a temporal agent, generated during the new schema introduction process and allows message passing to other agents. An explicit representation of connectors enables an automatic application of temporal agents message passing. The edges of a temporal dependency graph designate trigger and request relationships between elements.

Let $TDG_i = (V, E)$ be the temporal dependency graph of a temporal agent TAG_i . TDG_i is a directed graph, such that:

$$V = OP \cup EV \cup CO, \text{ where:}$$

OP is the set of all the operations in s_i . An operation node is generated for two types of operations; *direct update operations* that denote update operations for properties that are explicitly updated (e.g., `Subscriber.Address`) and *dependency preservation operations* inferred from a dependency definition (e.g., `Distributor.Commission`).

EV is the set of all the events in s_i .

CO is the set of all the connectors in s_i . $CO = SCO \cup RCO$, where SCO is a set of sending connectors, and RCO is a set of receiving connectors.

$E = TR \cup RE$, where:

TR is a triggering edge. It connects an element $op_i \in (OP \cup EV \cup RCO)$ with an element $op_j \in (OP \cup SCO)$.

Let p be a property. A triggering edge $\langle op_i, op_j \rangle$ is generated if one of the following six conditions holds:

- 1) $p \in UPDATE-SET(op_i)$, i.e., op_i updates an instance of p and $p \in TRIGGER-SET(op_j)$. For example, according to Fig. 3, operation (o5) updates Number-of-Subscribers, which triggers (o1). Therefore, a triggering edge connects (o5) with (o1).
- 2) op_i is an event that triggers op_j , i.e., $op_i \in TRIGGER-SET(op_j)$. For example, operation (o8) is activated as a result of the event Begin-of-Year. Consequently, a triggering edge connects the Begin-of-Year node with (o8).
- 3) $p \in TRIGGER-SET(op_i)$, op_j is a sending connector, and p is the source of op_j . For example, assume that at t' , operation (o8) was changed. As a result, a connector that models the effect of updating Distributor-Status using (o6), on (o8) as of t' is added to the two schema versions. A transaction that changes Distributor-Status during $[t' - 1year, t')$, requires a message passing, to enable the derived update of Annual-Commission. Assume further that Distributor-Status triggers (o8). Using this scenario, a triggering edge connects (o6) with a sending connector.
- 4) op_i is a receiving connector, and op_j is the destination of op_j . Using the example given above, a triggering edge connects a receiving connector with (o8).
- 5) op_j is a sending connector, and op_i is the source of op_j . For example, assume a new schema as a result of change in the domain of Salary, starting t'' . As a result, a connector that models the effect of updating Annual-Commission during $[t'', t'' + 1year)$ is added to the two schema versions. Using this scenario, a triggering edge connects (o8) with a sending connector.
- 6) $p \in UPDATE-SET(op_j)$, op_i is a receiving connector, and p is the destination of op_j . Under the scenario given above, a triggering edge connects a receiving connector with (o8).

RE is a request edge. It connects an element $op_i \in (OP \cup EV \cup RCO)$ with an element $op_j \in (OP \cup SCO)$.

Let p be a property. A request edge $\langle op_i, op_j \rangle$ is generated if one of the following two conditions hold:

- 1) $p \in UPDATE-SET(op_i)$, i.e., op_i updates an instance of p , and $p \in REQUEST-SET(op_j)$. For example, consider the operation (o8) given in Fig. 3. $Distributor-Status \in REQUEST-SET((o8))$ and $Distributor-Status \in UPDATE-SET((o6))$. As a result, a request edge connects (o6) with (o8).

- 2) $p \in REQUEST-SET(op_j)$, op_i is a receiving connector, and p is the destination of op_j . For example, assume that at t' , operation (o8) was changed. As a result, a connector that models the effect of updating Distributor-Status, using (o6) on (o8) as of t' is added to the two schema versions. A transaction that may involve a change in Distributor-Status in $[t' - 1year, t')$, requires a message passing, to enable the derived update of Annual-Commission. Using this scenario, a request edge connects (o6) with a sending connector.

The space complexity of the graph is bounded by $O(|OP| + |EV| + |CO|)$, i.e., the number of metadata entities (operations, events and connectors). According to previous analysis, and under the assumption that $|EV| \leq |OP|$, the space complexity of the graph is bounded by $O(\max_{i=1}^n (|oper_i|)^2)$. The number of nodes and edges of the graph reflects the number of operations, events and connectors, and the relationships among them, and not the number of objects in the database. Thus, the space complexity of the graph is quadratic proportional to the size of the metadata entities, which is normally a comparatively low overhead relative to the size of the database. The complexity calculation is not dependent on the relationships between the number of metadata elements and the number of data elements. Consequently, even in extreme cases (e.g., initialization of a database) where the number of metadata elements is of the same order as the number of data elements, there is no change in the complexity calculation.

Fig. 5 presents a partial example of the temporal dependency graph of the distributors' case study. The temporal dependency graph contains two types of operations. The operations (o1), ..., (o9) are inferred from the derivations presented in Fig. 3, and the direct update operations (o'1)-(o'10), shown below:

- (o'1) updates a Zipcode variable.
- (o'2) updates an Expiration-Date variable.
- (o'3) updates a Subscriber's-Limit variable.
- (o'4) updates a Medium-Lower-Bound variable.
- (o'5) updates a High-Lower-Bound variable.
- (o'6) updates a Distributor-Rank variable.
- (o'7) updates a High-Commission-Rate variable.
- (o'8) updates a Medium-Commission-Rate variable.
- (o'9) updates a Low-Commission-Rate variable.
- (o'10) updates a Total-Dividend variable.

The receiving connector rco_1 represents a change in the operation (o8) that calculates the annual commission. The operation may need data that exists at a time element that is monitored by a different agent. Therefore, connectors are added to the temporal dependency graphs.

The generation of the temporal dependency graph is inferred from the schema and the dependency definitions. The time-complexity of generating the temporal dependency graph is bounded by $O(\max(|V| + |E|), (|V| \cdot |E|))$ [21], where $O(|V| \cdot |E|)$ is the time complexity of generating the trigger and request relationships. In terms of the number of

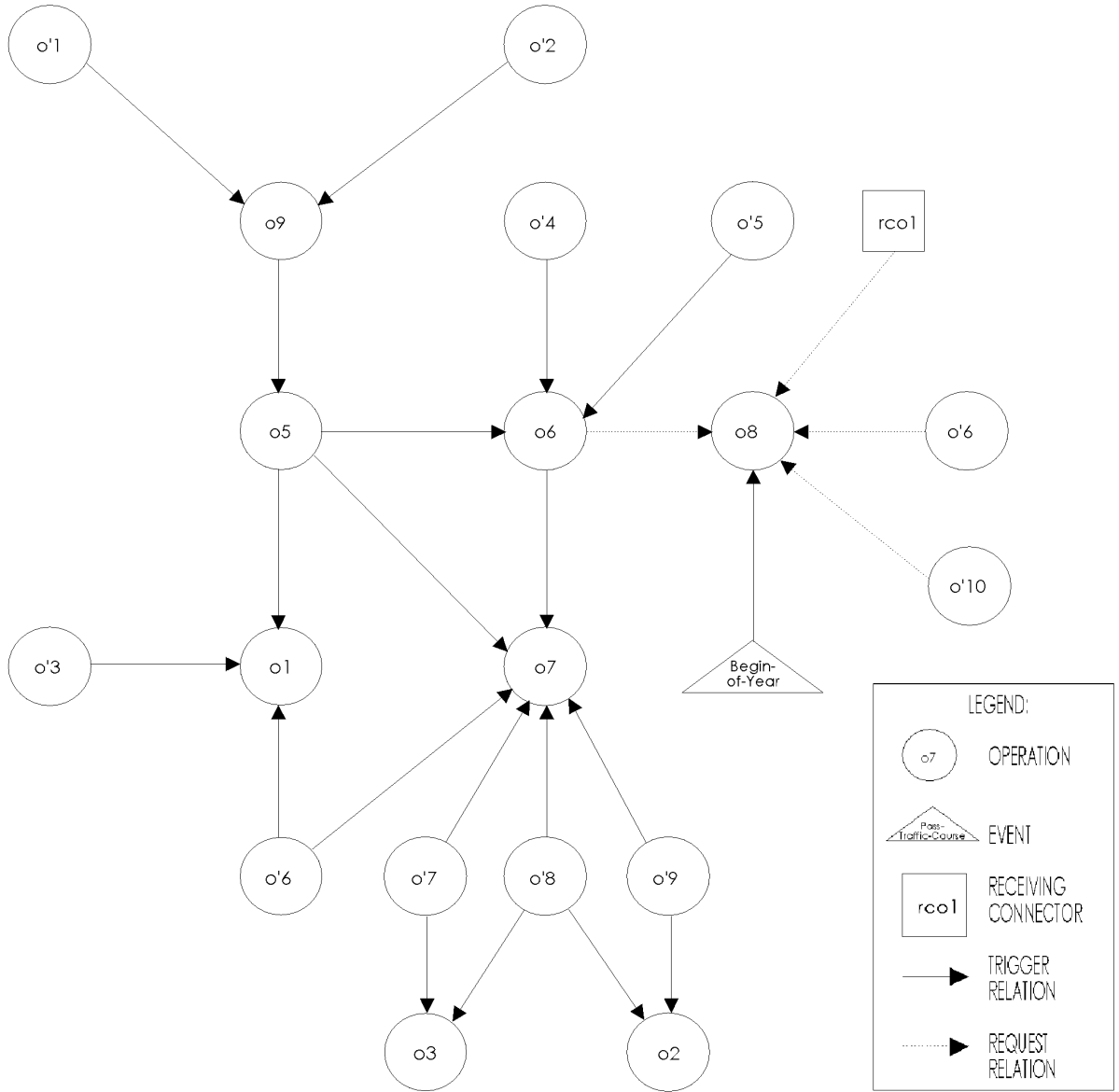


Fig. 5. A partial temporal dependency graph of the distributors' case study.

nodes in the graph, the algorithm is bounded by $O(|V|^3)$, where $|V|$ denotes the number of metadata entities (properties, operations, connectors and events). The generation of the temporal dependency graph is required only at the application's initialization and changes may be done in an incremental fashion.

In addition to the temporal dependency graph, the update algorithm utilizes the element subgraph of all the nodes representing events, direct update operations and receiving connectors. Each subgraph contains the transitive closure of a single node. The derivation of these subgraphs can be determined in a preprocessing phase. An *element subgraph* of a node v_i is a directed graph $ESG_i(v_i) = (V, E)$:

$$V = \{v \mid \exists \text{ a path } \langle v_i, \dots, v \rangle \in TDG_i\}.$$

$$E = \{\langle v_j, v_k \rangle \mid \exists \text{ a path } \langle v_i, \dots, v_j, v_k \rangle \in TDG_i\}.$$

Fig. 6 represents an element subgraph of operation (o'1) that updates the Zipcode variable.

The execution process is monitored by assigning weights to each TDG_i , as defined in Section 3.3. For each transaction, the edges of TDG_i receive different weights. TDG_i is built from the element subgraphs of all the operations, events and receiving connectors that are initiated by the transaction.

3.3 The Runtime Execution Process

The transaction manager receives as an input a set of direct update operations and event signals o_1, \dots, o_n . The direct update operations are given by the user, and the events can be either signaled by the user, or detected by event detectors that receive information from external sensors. Each update operation is decomposed by the transaction manager to primitive operations. A primitive operation is a tuple $\langle operation, variable-name, value, t_v \rangle$, where *operation* is either an update operation of a single variable, or a signal

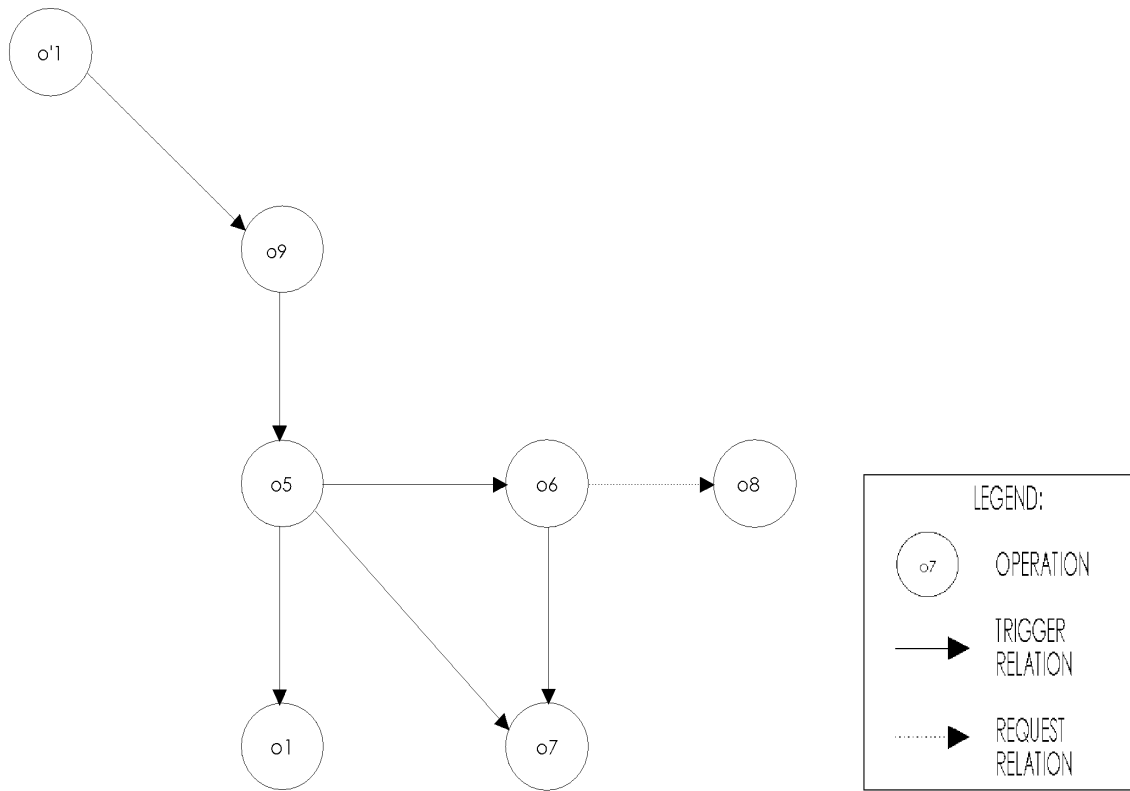


Fig. 6. An element subgraph of ($o'1$).

operation. The primitive operations are sent to the execution manager that assigns each primitive operation to a temporal agent. The update process is a two phase process that uses the temporal dependency graph as an executable data structure to monitor the update process, as described in Sections 3.3.1 and 3.3.2.

3.3.1 Phase I: The Determination of the Transaction's Range of Effect

Each temporal agent TAG_i should decide upon the transaction's range of effect in its associated time region on the valid time axis. Each TAG_i updates its temporal dependency graph TDG_i for future monitoring of the potential activation, requests, and updates.

Fig. 7 presents the algorithm for determining the transaction's range of effect. The algorithm is performed independently by each of the agents that are involved in the transaction. The algorithm updates the weights of the temporal dependency graph (TDG_i). An edge has a positive weight if the source operation of the edge should be activated during the transaction, and has not completed its activities yet. Each edge is initialized to have a weight of 0 (line 2). The weights change according to the element subgraphs of all the operations that are initiated by the transaction. If an operation o is one of the primitive operations of the transaction, then its successors will be activated during the transaction. Hence, the triggering edges that are related with these operations receive a positive weight. $TDG_i \uplus ESG_i(o)$ (line 5) stands for an increase of one to the weight of an edge in TDG_i for each edge that exists in $ESG_i(o)$.

If a sending connector is a node of one of the element subgraphs that are relevant to the transaction, then messages should pass among agents (lines 12-15 and lines 21-24), to signal the possible activation of a subgraph that its root is a connector. Each agent that receives a message of possible activation adds the effect of the subgraph of the receiving connector to the temporal dependency graph.

For example, Fig. 8 presents an example of messages passing between two agents, in the phase of determining the transaction range of effect. In this figure, a partial shema of the distributors case-study based on the dependencies of Fig. 3 is presented. At t' , operation ($o8$) was changed. As a result, a connector that models the effect of ($o6$) on ($o8$) as of t' is added to the two schema versions. A transaction that may involve a change in Distributor-Status in $[t' - 1\text{year } t')$ requires a message from $sco1$ to $rco1$ to enable the derived update.

In lines 17-28 the weight of each of the request edges is calculated. A request edge $\langle v_i, v_j \rangle$ receives the value of 1 if there exists a trigger edge to v_i and a trigger edge from v_j that should be triggered during the transaction, or if there exists a trigger edge to v_i and a request edge from v_j to a sending connector v_l . In the latter case, the receiving connectors that are the destination of v_l are signaled (lines 23-24).

3.3.2 Phase II: The Execution of Operations

In the execution phase, each temporal agent TAG_i uses its temporal dependency graph TDG_i to determine the flow of operations. Each operation can be either activated by a direct update or triggered.

Determine-Transaction-Range-of-Effect:

```

Input:       $u_1, \dots, u_n$  /* primitive operations */
Output:     $TDG_i$  /* A temporal dependency graph */
            $ESG_i(o_1), \dots, ESG_i(o_m)$  /* an element subgraph of each source operation in  $TDG_i$  */
1         For each edge  $\langle v_i, v_j \rangle \in TDG_i$  do:
2           weight( $\langle v_i, v_j \rangle$ ) := 0
3         For i := 1 to n do:
4           If  $u_i$ .operation-type = "signal" then:
5              $TDG_i := TDG_i \uplus ESG_i(u_i.name)$ 
           /* increase by 1 the weight of an edge in  $TDG_i$  for each triggering edge that exists
           in  $ESG_i(u_i.name)$  */
6           else:
7             Find  $o_j$  such that  $u_i.name \in UPDATE-SET(o_j)$ 
8              $TDG_i := TDG_i \uplus ESG_i(o_j)$ 
9           end If
10        end For
11        For each edge  $\langle v_i, v_j \rangle \in TR(TDG_i)$  do:
12          If weight( $\langle v_i, v_j \rangle$ ) = 1  $\wedge v_j \in SCO(TDG_i)$  then:
13            For each  $rco \in destination(v_j)$  do
14              Send Signal-Connector-Message ( $rco$ )
15            end If
16          end For
17          For each edge  $\langle v_i, v_j \rangle \in RE(TDG_i)$  do:
18            If  $\exists \{\langle v_k, v_i \rangle, \langle v_j, v_l \rangle\} \subseteq TR(TDG_i) \mid weight(\langle v_k, v_i \rangle) = weight(\langle v_j, v_l \rangle) = 1$  then:
19              weight( $\langle v_i, v_j \rangle$ ) := 1
20            else:
21              If  $\exists \langle v_k, v_i \rangle \in TR(TDG_i) \wedge \exists \langle v_j, v_l \rangle \in RE(TDG_i) \mid weight(\langle v_k, v_i \rangle) = 1 \wedge v_l \in SCO(TDG_i)$ 
                then:
22                weight( $\langle v_i, v_j \rangle$ ) := 1
23                For each  $rco \in destination(v_j)$  do:
24                  Send Signal-Connector-Message ( $rco$ )
25                else weight( $\langle v_i, v_j \rangle$ ) := 0
26              end If
27            end If
28          end For

```

Fig. 7. The determine-transaction-range-of-effect algorithm.

The execution phase starts with the activation of each of the operations and events that were sent to the agent. The activation of an operation entails retrieval of state-elements, invocation of functions, and calculation of new values to be inserted to the database. Upon completion of these activities, the database is updated, and all the outgoing edges of the operation are triggered. The activation of an event results in triggering its outgoing edges.

An operation is triggered when one of its predecessors in the TDG_i is terminated. A triggered operation reduces the weight of the triggering edge, and consults the TDG_i to assess its status. If there are no more ingoing edges to be triggered, i.e., all ingoing edges have a zero weight, then the operation is activated. Otherwise, the operation should wait for the next triggering. This mode of operation guarantees that each activated operation is scheduled only once.

An operation that should be triggered by another schema version, should wait until it receives a message from the sending connector's agent. A message is sent once the execution reaches a sending connector. A message is sent to the receiving connectors in the destination of the

sending connector, and enables the triggering of the receiving connector's outgoing edges.

For example, a transaction that signals the event **Begin-of-Year** and decreases the **Medium-Lower-Bound** for the period $[t' - 1year, t')$ was issued on t' . As a result, following Fig. 8 and Fig. 5, **(o'4)** is activated and updates **Medium-Lower-Bound**. **(o6)** is activated next, and updates the **Distributor-Status**. **(o8)** of another schema should receive the **Distributor-Status** new values. Thus, the sending connector *sco1* is triggered, and a message is sent to *rco1* of the receiving agent, notifying that the update of **Distributor-Status** is now complete. At this stage, **(o8)** of the other schema can be activated.

4 THE MODEL'S PROPERTIES

In this section we present the properties of the multiagent model. We discuss four issues, some are inherited from the properties of the PARDES model and some are unique to the temporal extension. Section 4.1 examines the model in view of the initial motivation posed in Section 1. The notion of database consistency is discussed in Section 4.2. Section

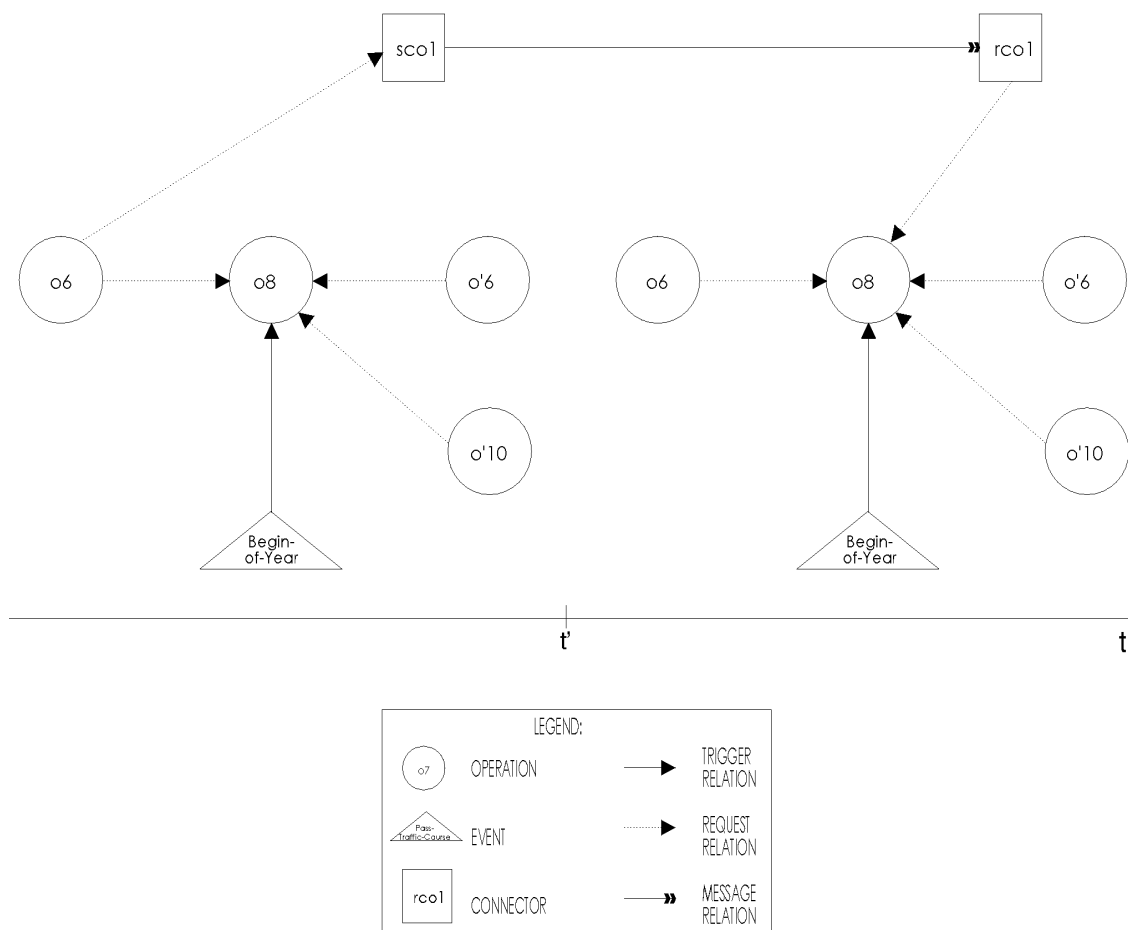


Fig. 8. An example of messages passing.

4.3 discusses operation termination and confluence along temporal graphs, and performance and feasibility issues are discussed in Section 4.4.

4.1 Revisiting The Initial Motivation

Section 1 outlined an initial motivation to construct a high-level language and execution model for temporal dependencies and posed two requirements for the temporal dependency model. These requirements are fulfilled using the following three properties.

- The database enables the concurrent maintenance of different schema versions. Each version is valid during a collection of time-points within the database's active time domain. This feature is vital to satisfy the **evolution of dependencies over time** requirement.
- The model supports valid time retroactive and proactive updates. An update operation may be issued for values that refer to the past or the future, with respect to the issuing transaction's viewpoint. In particular, it is possible to update a value for a time point in which a different schema version is valid. In this case the update is executed in the context of the valid schema version and not in the context of the current one. This feature is vital to satisfy the **use of temporal operators within the definition of data dependencies**.

- The temporal dependency language supports inter-version references. This property extends the utilization of schema versioning to the execution of dependencies, thus a change in a value that is valid in one schema version may entail the change of another value that belongs to another schema version.

$$x := \langle a, t_{v_2} \rangle + \langle b, t_{v_3} \rangle \text{ valid-in } t_{v_1} \quad (4)$$

For example, in the dependency given in Expression 4, a modification of a value of a during t_{v_2} entails the recalculation of the value of x during t_{v_1} that may belong to a different schema version. This feature integrates both requirements.

The update model presented in this paper enables an efficient execution of these features, making it a feasible model as elaborated in Section 4.4.

4.2 Database Consistency

Most DBMS tools supply structural consistency facilities such as type checking, referential integrity and cardinality enforcement. The data-dependency concept can be viewed as an additional means of expressing database's consistency requirements. A database is consistent with respect to a data dependency, if each data dependency is satisfied for all the instances of the relevant data elements that participate in it. Data-driven dependencies issue a *continuous*

consistency requirement, i.e., consistency requirement that should be satisfied in any stable state⁷ of the database. Event driven dependencies issue a *discrete consistency requirement*, i.e., consistency requirement that should be satisfied in any stable state of the database following directly an associated event detection. Temporal databases add a *snapshot consistency requirement*, whereby a value *val* that is valid at time-point *t* is consistent with respect to the valid schema version and data dependencies that are valid in *t*. Temporal data dependencies add a *temporal consistency requirement*, whereby a value *val'* with valid time *t'_v* that is derived from a value *val* with a valid time *t_v* is consistent with respect to the value of *val* and the dependency that derives *val'*. All of the dependencies presented in Fig. 3 requires a snapshot consistency. The dependencies (d4) and (d8) require temporal consistency as well.

A database is consistent if it satisfies snapshot consistency and temporal consistency in addition to the continuous or discrete consistency, depending on the dependency definition. For example, consider a database with three properties *x*, *a*, and *b* of the same class, and the two dependencies:

- (δ_1) $x := \langle a, t - 3 \rangle + \langle b, t - 3 \rangle$ valid-in *t*
- (δ_2) $x \leq 100$

An instance α of this class is updated, and *a* is set to be 38 during [Jan 1994, Feb 1994). The value of *b* is 70 during [Jan 1994, Feb 1994). The consequence of (δ_1) is the assignment of the value 108 to *x* during [Apr 1994, May 1994). On the one hand, if the new value of *x* is persisted in the database, the snapshot consistency is violated, since the value violates (δ_2) during [Apr 1994, May 1994). On the other hand, if the new value of *x* is not persisted, then the snapshot consistency is satisfied at each time-point *t*. However, the temporal consistency is violated, since (δ_1) is not satisfied during [Apr 1994, May 1994).

4.3 Operations Termination and Confluence

One of the main issues in systems that support data dependencies, such as production systems or active databases, is the issue of confluence and termination [3]. As shown in [2], the execution of an acyclic dependency graph is guaranteed to terminate, while the execution of a cyclic dependency graph is not. In the temporal data-dependency context, detection of cycles should be done both at the schema level and at the temporal level. For example, consider the following two assertions of a strategic decision support system.

- 1) If there is no threat of war in time τ_1 then the number of ballistic missiles can be reduced by *X* units during the interval $[\tau_1, \tau_1 + 1\text{year})$. Otherwise, the number should be increased by *X* units in the same period.
- 2) If it is known that the number of missiles in τ_2 will be less than *Y* units, then a threat of war exists as of $\tau_2 - 18$ months. Otherwise, there is no threat of war.

On January 1994, the number of ballistic missiles is updated to be *Z*, where $Z - X < Y < Z$. The information of “no

threat of war” would cause an infinite temporal cycle, where a threat of war is inferred and refuted in a cyclic fashion.

Temporal cycles may result from operations that refer to different time points, possibly among different schema versions, and can be detected by using a global view of the system, as follows.

Let $\text{TDG}_i = (V_i, E_i)$ ($1 \leq i \leq n$) be a collection of temporal dependency graphs in a database *D*. The *unified temporal dependency graph* $\text{UTDG} = (V, E)$ is a directed graph, where:

$$V = \bigcup_{i=1}^n V_i$$

$$E = \left(\bigcup_{i=1}^n V_i \right) \cup E_c$$

E_c is a set of *connecting edges* that connects each sending connector with its receiving connectors.

The unified temporal dependency graph represents the global dependencies of the database *D*. Using conventional graph analysis, cycles in the unified dependency graph imply the existence of temporal cycles in the multiversion system. Following [2], it is easy to show that:

- 1) If the unified temporal dependency graph has no cycles, then the multiversion update process would terminate in a finite number of steps.
- 2) If the multiversion update process does not terminate in a finite number of steps, then the unified temporal dependency graph has cycles.

The unified temporal dependency graph is incrementally updated with every schema version introduction, and temporal cycles are sought and detected.

4.4 Feasibility and Performance

In this section, we summarize the analysis of the time and space complexity overhead of the model (Section 4.4.1), and discuss some of its optimization features, namely update minimality, incremental updates and parallel processing (Section 4.4.2).

4.4.1 Overhead

The temporal dependency graph is an executable data structure that contains metadata elements. Hence, the additional space complexity of this model is proportional to the number of metadata elements. It is assumed that the size of the metadata is much smaller than the size of data and does not change the magnitude of the space complexity relative to any schema knowledge representation.

The number of schema versions is assumed to be relatively small, since it is assumed that metadata changes are grouped into versions [13]. Consequently, the number of interacting agents is relatively small.

The time complexity of the graph generation is $O(V^3)$, where *V* stands for the number of nodes that represent metadata elements. This process is performed during the initialization of the first schema version. Although subsequent schema versions revise a small part of the schema elements, the worst case of each revision is still $O(V^3)$.

Our complexity analysis as well as our experimentation with a prototype implementation encourage us to believe

7. A stable state is a state that is created when a transaction commits [18].

that the time and space complexity are not substantial. Due to the high-level language abstractions, the development and maintenance costs are likely to be substantially reduced with respect to development using conventional tools. The overall effect of the combination of required overhead and supplied optimization features is likely not to be substantially worse and in some cases even improve the performance of applications that have the same functionality and are developed using conventional tools. For example, we compared the following three possible strategies for handling schema versioning.

- 1) The database uses a single "active" schema at all valid time points. Using this method, the active schema is normally the schema that is the result of the last schema modification. This method supports schema evolution, by allowing changes in the schema level, but does not support schema versioning [32]. It should be noted that this strategy does not meet the requirements posed in this paper.
- 2) The database supports schema versioning by handling each schema version separately, in a sequential manner. This method uses a simple update algorithm, with no need for communication protocols.
- 3) The database supports a multiagent model in which the parallel processing decreases the computation time, but the synchronization, coordination, and communication among agents generate a performance overhead.

We have conducted a simulation that has evaluated the total performance of a transaction as a function of the number of operations in a transaction, and the method of applying schema versioning. At each run of the simulation, the communication overhead was assumed to be a constant. Fig. 9 presents an instance of this simulation set, with an overhead of 12 percent. A set of 100 transactions, each one consists of a similar number of operations, was tested on the three strategies of schema versioning support. One with no schema versioning support, the other with a sequential support of two schema versions that halved the application time domain, and the last with a multiagent model that supports two schema versions that halved the application

time domain. In all cases, the application time domain was an interval of the form $[0, v)$, where $v = 300$. Each update operation updated a variable with a valid time $[t_s, t_e]$, such that $t_s \sim U[0, v]$ and $t_e \sim U[t_s, v]$. The update operations were designed to succeed in all of the time regions; thus, no failure overhead should be considered and the only trade-off in a multiagent model is the cost of communication vs. the benefit of parallel execution.

According to the results presented in Fig. 9, the use of sequential handling of multischema versions has the worst performance of all three strategies. The interesting result of this simulation is that when the number of operations in a transaction is four or above, and the communication overhead is 12 percent, the performance of the schema versioning with a multiagent option model is better than the non-versioning one.

4.4.2 Optimization

At runtime there are several features that improve the application performance relative to ad hoc designed applications. These features are update minimality, incremental updates, and parallel processing:

- **Update minimality:** Update minimality is achieved when the number of update operations is minimal, i.e., each update operation is unavoidable and none of the operations is redundant. This property is inherited from the PARDES execution model [17] and is based upon the fact that the execution model accumulates relevant triggers and activates each update operation only when all its predecessors in the graph that are updated in the transaction have been activated. This accumulation eliminates the redundancy update problem that is common in data-driven systems [30].
- **Incremental updates:** A substantial amount of computation can be performed in an incremental fashion. For example, in the data dependency $y := \text{sum}(x)$, a modification in one of the relevant instances of x does not require the recalculation of the sum, which can be done incrementally. A thorough discussion of

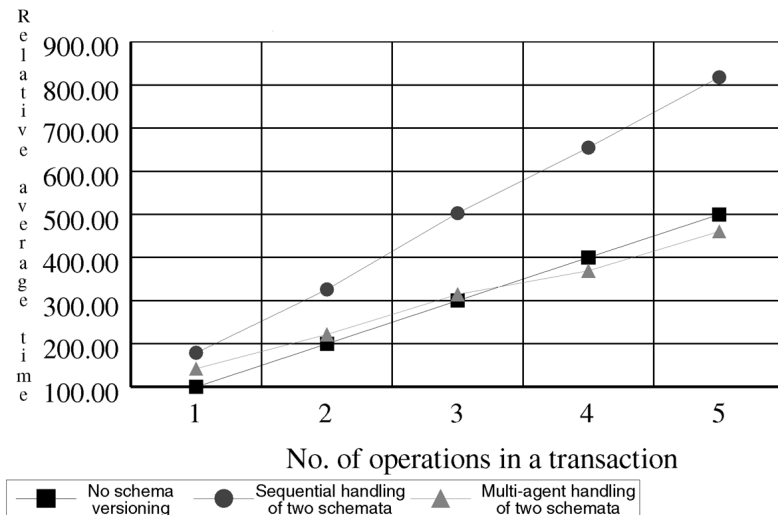


Fig. 9. The average elapsed time of transactions for the three schema versioning handling methods.

incremental computation of data dependencies can be found in [34].

- **Parallel processing:** The partition of the application time domain enables the support of parallel processing. The contribution of parallel processing vs. the communication overhead has been investigated by simulation [23]. As stated above, the major cost of parallel processing is the overhead generated by the need to coordinate and to transfer information among agents. The communication overhead in Fig. 9 was assumed to be a constant (12 percent). The simulation concluded that for two agents it is beneficial to use the multiagent model regardless of the communication overhead, for transactions with four operations or more. For three agents, the multiagent model is preferred for transactions with four operations or more, and for a communication overhead of up to 15 percent. It is worth noting, that the alternative method is not to support schema versioning at all. Thus, these results show the benefits of using the multiagent model due to time considerations only, while increasing the capabilities of the supporting database. A more detailed discussion of the simulation is presented in [23].

5 RELATED WORK

The linguistic and execution aspects of data dependencies have been thoroughly investigated using both event driven techniques [7] and data-driven techniques [34], [30], [18]. A comparison among different approaches for executable specifications of data dependencies can be found in [22]. While existing models support various types of data dependencies, none of them accommodate evolution of data dependencies over time and the use of temporal operators within the definition of data dependencies, as required in this paper.

Several works in the temporal database research area were published on topics related to the combination of temporal aspects and rules, e.g., [33], [46], [14], and [40]. These models possess temporal capabilities, but they neither support explicit data dependencies nor do they enable retroactive and proactive updates in a database that supports schema versioning. Temporal query languages (e.g., [44]) support time characteristics such as valid time and transaction time as part of the language, and the use of defaults in defining these characteristics, in case the user does not provide them. However, existing temporal query languages offer a weaker model of temporal data dependencies than presented in this paper.

Conventional active databases (e.g., HiPac [7], Ode [1], Starburst [47], and Sybase [25], [43]) support temporal functionality up to a limited extent by supporting only limited types of temporal events. Therefore, the use of existing graph based tools (e.g., [2]) cannot capture temporal semantics. For example, the use of connectors based on valid time as defined in temporal data dependencies is disabled since these models do not provide a support for history maintenance. It is worth noting that despite the fact that, in general, the maintenance of data dependencies can be performed

using active database systems, these two disciplines do not overlap. On the one hand, the required semantics of dependencies is not easily expressible in active databases [18]. On the other hand, the support of data dependencies does not require all the capabilities of active databases.

Graph-based data models (e.g., [26], [36]) serve as an alternative to conventional data models. While many graph-based data models represent data as well as metadata, this work uses graph techniques for metadata modeling solely, and therefore many of the techniques that are given in these models are redundant. Most of the graph based data models (e.g., LDM [35], Graphlog [10], and GOOD [27]) use a single digraph. Hence, the presentation of several schema versions is awkward. While the Hypernode model [36] make use of a finite set of digraphs, it is not evident that the connectivity among these digraphs is expressive enough to define connectors. To the best of our knowledge, none of the graph-based data models have an explicit notion of time.

Transaction data-flow graphs (e.g., [15], [39]) can represent data precedence as a result of read/write conflicts in a transaction. Our model can express more complicated relationships among general types of operations, using temporal semantic knowledge to ensure a correct update of the database under schema versioning.

An architecture for implementing temporal integrity constraints by compiling them into a set of active DBMS rules was suggested in [8]. This approach is limited and do not support rules that update the temporal database valid time history. In addition, the paper assumes a single set of temporal constraints. Hence, no support of schema versioning is discussed in this work.

Real time databases [38] enforce temporal consistency and handle multiversion data [45]. These models assume a database with no schema versioning, and their treatment of multiversion data is strict. For example, in [45], it is assumed that a new data value replaces an older one. While this assumption is possibly true in some real time databases, it is not valid for models of decision support, supported by the model presented in this paper.

6 CONCLUSION

This paper introduces a model for an update process of a database with temporal data dependencies and schema versioning. This model extends the useful notion of data dependencies to consist of temporal knowledge. Several techniques such as the use of temporal dependency graph and the parallel execution model are used to optimize the update execution. The paper presents the following novel features:

- High level support of schema versioning. This model supports the concurrent use of different schema versions and allows high level abstractions to update data in several schema versions within a single operation. This approach is extendible to cooperative systems in which different users possess different versions of the schema. The alternative way employed in current systems is to perform manually all the implications upon the distinct versions.

- A temporal dependency language that embeds the temporal dimension in the dependency language.
- The view of schema history as a collection of temporal regions, each controlled by a temporal agent, applies the ideas of parallel processing to temporal databases.

We have implemented a prototype of the temporal data-dependencies model using the X86 platform. Our initial experimentation with a variety of applications is encouraging from the performance perspective. However, a definite conclusion using a large sample of applications is a topic for further research. Additional further research will deal with exception handling in the proposed environment and hypothetical schema versioning to support possible worlds. We shall research the extension of the model to a distributed environment with no global control and test the applicability of this model on change management models.

ACKNOWLEDGMENTS

The basic work of combining the active and temporal technologies has been done in collaboration with Arie Segev. We thank the reviewers for their helpful comments that helped us to improve the structure and readability of the paper. This research was done while both authors were at the Israel Institute of Technology (Technion). Opher Etzion's work was supported by the fund for the promotion of research at the Technion.

REFERENCES

- [1] R. Agrawal and N.H. Gehani, "Ode (Object Database and Environment): The Language and the Data Model," *Proc. 1989 ACM SIGMOD Conf. Management Data*, Portland, Ore., June 1989.
- [2] A. Aiken, J. Widom, and J.M. Hellerstein, "Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism," *Proc. ACM SIGMOD*, pp. 59-68, June 1992.
- [3] E. Baralis, S. Ceri, and J. Widom, "Better Termination Analysis for Active Databases," *Proc. First Int'l Workshop Rules Database Systems*, pp. 163-179, Aug. 1993.
- [4] J. Blakeley, N. Cobourn, and P.A. Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," *ACM Trans. Database Systems*, vol. 14, no. 3, pp. 322-368, 1989.
- [5] D. Botzer and O. Etzion, "Optimization of Materialization Strategies of Derived Data-Elements," *IEEE Trans. Knowledge and Data Eng.*, vol. 8, no. 2, pp. 260-272, 1996.
- [6] S. Chakravarthy and D. Mishra, "An Expressive Event Specification Language for Active Databases," *Data and Knowledge Eng.* vol. 13, no. 3, Oct. 1994.
- [7] U.S. Chakravarthy, "Rule Management and Evaluation: An Active DBMS Perspective," *ACM SIGMOD Record*, vol. 18, no. 3, pp. 20-28, Sept. 1989.
- [8] J. Chomicki and D. Toman, "Implementing Temporal Integrity Constraints Using an Active DBMS," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 4, pp. 566-581, Aug. 1995.
- [9] CODASYL, Database task group report, 1971.
- [10] M.P. Consens and A.O. Mendelzon, "Graphlog: A Visual Formalism for Real Life Recursion," *Proc. ACM SIGACT-SIGMOD-SIGART Symp. Principles Database Systems*, Nashville, Tenn., Apr. 1990.
- [11] D.J. Dewitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Comm. ACM*, vol. 35, no. 6, pp. 85-98, 1992.
- [12] K.R. Dittrich and S. Gatzin, "Time Issues in Active Database Systems," *Proc. Int'l Workshop Infrastructure Temporal Databases*, Arlington, Texas, June 1993.
- [13] D. Dori, A. Gal, and O. Etzion, "Temporal Active Databases: A Key to Computer Integrated Manufacturing," *Int'l J. CIM*, vol. 9, no. 2, pp. 89-104, 1996.
- [14] M.L. Edara and S.K. Gadia, "Updates and Incremental Recomputation of Active Relational Expressions in Temporal Databases," *Proc. Int'l Workshop Infrastructure for Temporal Database*, Arlington, Texas, June 1993.
- [15] M.H. Eich, "Graph Directed Locking," *IEEE Trans. Software Eng.* vol. 14, no. 2, p. 133, Feb. 1988.
- [16] O. Etzion, "Active Interdatabase Dependencies," *Information Sciences*, vol. 75, pp. 133-163, 1993.
- [17] O. Etzion, "PARDES—A Data-Driven Oriented Active Database Model," *ACM SIGMOD Record*, vol. 22, no. 1, pp. 7-14, Mar. 1993.
- [18] O. Etzion, "The Reflective Approach for Data-Driven Rules," *Int'l J. Intelligent and Cooperative Information Systems*, vol. 2, no. 4, pp. 399-424, Dec. 1993.
- [19] O. Etzion, A. Gal, and A. Segev, "Retroactive and Proactive Processing," *Proc. Research Issues Data Eng.—Active Database Systems*, pp. 126-131, Feb. 1994.
- [20] S.K. Gadia, "The Role of Temporal Elements in Temporal Databases," *Data Eng. Bull.*, vol. 7, pp. 197-203, 1988.
- [21] A. Gal, "TALE—A Temporal Active Language and Execution Model," PhD thesis, Technion (Israel Inst. of Technology), Technion City, Haifa, Israel, May 1995; available at WWW <http://cs.toronto.edu/avigal>
- [22] A. Gal and O. Etzion, "Maintaining Data Driven Rules in Databases," *Computer*, vol. 28, no. 1, pp. 28-38, Jan. 1995.
- [23] A. Gal and O. Etzion, "A Parallel Execution Model for Updating Temporal Databases," *Int'l J. Computer Systems Science and Eng.*, vol. 12, no. 5, pp. 317-327, Sept. 1997.
- [24] A. Gal, O. Etzion, and A. Segev, "Representation of Highly Complex Knowledge in a Database," *J. Intelligent Information Systems*, vol. 3, no. 2, pp. 185-203, Mar. 1994.
- [25] A. Gorelik, Y. Wang, and M. Deppe, "Sybase Replication Server," *ACM SIGMOD Record*, vol. 23, no. 2, p. 469, 1994.
- [26] R.L. Griffith, "Three Principles of Representation for Semantic Networks," *ACM Trans. Database Systems*, vol. 25, no. 9, p. 666, Sept. 1982.
- [27] M. Gyssens, J. Paredaens, and D. Van Gucht, "A Graph-Oriented Object Database Model," *IEEE Trans. Knowledge and Data Eng.*, vol. 6, no. 4, p. 572, 1994.
- [28] M. Hammer and D. McLeod, "Data Base Description with SDM: A Semantic Data Base Model," *ACM Trans. Database Systems*, vol. 6, no. 3, 1981.
- [29] E.N. Hanson, "A Performance Analysis of View Materialization Strategies," *Proc. SIGMOD*, pp. 440-453, June 1987.
- [30] S. Hudson and R. King, "CACTIS: A Database System for Specification Functionality Defined Data," *Proc. IEEE OOBDS Workshop*, pp. 26-37, Sept. 1986.
- [31] C.S. Jensen, J. Clifford, S.K. Gadia, A. Segev, and R.T. Snodgrass, "A Glossary of Temporal Database Concepts," *ACM SIGMOD Record*, vol. 21, no. 3, pp. 35-43, 1992.
- [32] C.S. Jensen et al., "A Consensus Glossary of Temporal Database Concepts," *ACM SIGMOD Record*, vol. 23, no. 1, pp. 52-63, 1994.
- [33] M.R. Klopprogge and P.C. Lockmann, "Modeling Information Preserving Databases; Consequences of the Concept of Time," *Proc. Int'l Conf. VLDB*, Florence, Italy, 1983.
- [34] S. Koenig and R. Paige, "A Transformational Framework for the Automatic Control of Derived Data," *Proc. Seventh Conf. Very Large Data Bases*, Zaniolo and Delobel, eds., pp. 306-318, Morgan Kaufmann, Los Altos, Calif., Sept. 1981.
- [35] G.M. Kuper and M. Vardi, "On the Expressive Power of the Logical Data Model," *Proc. ACM SIGMOD Int'l Conf. Management Data*, pp. 180-187, Austin, Texas, ACM Press, 1985.
- [36] M. Levene and G. Loizou, "A Graph-Based Data Model and its Ramifications," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 5, pp. 809-823, Oct. 1995.
- [37] J. Mylopoulos and E. Yu, "Aligning Information System Strategy with Business Strategy: A Technical Perspective," *Proc. Int'l Workshop Next Generation Technologies and Systems*, Haifa, Israel, June 1993.
- [38] G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 4, Aug. 1995.
- [39] P.K. Reddy and S. Bhalla, "A Nonblocking Transaction Data Flow Graph Based Protocol for Replicated Databases," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 5, pp. 829-834, Oct. 1995.

- [40] N.L. Sarda, "HSQL: Historical Query Language," *Temporal Databases*, chapter 5, pp. 110-140, Benjamin/Cummings, Redwood City, Calif., 1993.
- [41] A. Segev and J.L. Zaho, "Data Management for Large Rule Systems," *Proc. VLDB*, 1991.
- [42] A. Sheth, M. Rusinkiewicz, and G. Karabatis, "Using Polytransactions to Manage Interdependent Data," *Trans. Models for Advanced Database Applications*, A. Elmagarmid, ed., chapter 14, Morgan Kaufmann, 1992.
- [43] A. Sistla and O. Wolfson, "Temporal Triggers in Active Databases," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 3, pp. 471-486, June 1995.
- [44] R. Snodgrass et al., "TSQL2 Language Specification," *ACM SIGMOD Record*, vol. 23, no. 1, pp. 65-86, Mar. 1994.
- [45] X.C. Song and J.W.S. Liu, "Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 5, pp. 786-796, Oct. 1995.
- [46] S.Y.W. Su and H.M. Chen, "A Temporal Knowledge Representation Model OSAM*/T and its Query Language OQL/T," *Proc. Int'l Conf. VLDB*, 1991.
- [47] J. Widom, "The Starburst Rule System: Language Design, Implementation, and Applications," *IEEE Bull. Technical Committee Data Eng.*, vol. 15, nos. 1-4, Dec. 1992.
- [48] G. Wiederhold, "From Data Engineering to Information Engineering," *Proc. Int'l Conf. Data Eng. (ICDE)*, Feb. 1994.

and the President's List for Highly Outstanding Achievements. In addition, he received the Miriam and Aaron Gutwirth Scholarship for three consecutive years. His research interests include active databases, temporal databases, cooperative information systems, metadata tools, and workflows. He has published 18 papers in journals and conference proceedings on these subjects.



Opher Etzion earned a BA degree in philosophy in 1980 from Tel-Aviv University and his PhD degree in computer science from Temple University in 1990. He is currently a research staff member at the IBM-Haifa Research Laboratory and on leave from the Technion, where he was the founding head of the information systems engineering area and graduate program. Prior to his graduate studies, he held professional and managerial positions in industry and in the Israel Air Force, receiving its highest award in 1982. His research interests include various issues related to intelligent information systems including high-level languages, active databases, reflective programming, and temporal databases. He has published more than 30 papers on these topics in professional journals and conference proceedings. He is a member of the editorial board of the *IIE Transaction Journal*; was a guest editor for the *Journal of Intelligent Information Systems* in 1994; is a member of the advisory committee of the Israeli systems analyst board; and is a member of the high commission of technical terms of the Hebrew Language Academy. He has served on many program committees, and has been a program chair and general chair of NGITS workshops; he served as the coordinating organizer of the Dagstuhl Seminar on Temporal Databases that was held in June 1997. He is a coeditor of the book *Temporal Databases: Research and Practice* that will be published in 1998 by Springer-Verlag. He is a member of the ACM, the IEEE Computer Society, the AIS, and the Israel Information Processing Association.



Avigdor Gal received his BSc degree (cum laude) in information systems engineering in 1990 and his DSc degree in 1995 from the Technion (the Israel Institute of Technology). He has been an assistant professor in the Department of MSIS at Rutgers University since the fall of 1997. From 1995 to 1997, he was a postdoctoral fellow and a member of the John Mylopoulos Group at the University of Toronto, Canada. During his studies, he was on the Dean's List for Outstanding Achievements