Extended Update Functionality in Temporal Databases

Opher Etzion
Technion - Israel Institute of Technology
Information Systems Engineering Department
Faculty of Industrial Engineering and Management
Haifa, 32000, Israel

Avigdor Gal
University of Toronto,
Department of Computer Science,
Toronto, Canada

Arie Segev
Haas School of Business, University of California and
Information & Computing Sciences Division,
Lawrence Berkeley Laboratory
Berkeley, CA 94720, USA

Abstract

This paper presents an extended update functionality in *temporal databases*. In temporal databases, the information is associated with several time dimensions that designate the validity of the information in the application domain as well as the database domain. The complexity of information, coupled with the fact that historical data is being kept in the database, requires the use of an update model that provides the user with high-level abstractions. In this paper we provide an enhanced schema language and an enhanced collection of update operation types that help the system designer and the user to cope with the added complexities of such a model. One of the major issues dealt with in this paper is the situation of *simultaneous values* of a single data item; this situation occurs when multiple values, valid at the same time, were assigned to a data item at different times over the database history. Unlike the fixed semantics in conventional and existing temporal database models, we provide a flexible mechanism to handle simultaneous values which also distinguishes between regular modifications and error corrections.

The extended update functionality is part of an update model that is currently being implemented in a prototype for a simulation project in a hospital's training center. Issues related to the implementation of this functionality in various data models are discussed. In particular, a mapping of the basic primitive operation types to TSQL2, and suggestions for its augmentation are provided.

Keywords: Temporal databases, Database updates, Simultaneous values, Decision Time, TSQL2

1 Introduction and motivation

Temporal databases enable the accumulation of information over time, and provide the ability to store different values of the same data item[1] with different time characteristics, thus enabling queries as if they were issued from past observation times (e.g. what was the known patient's situation at the time that a treatment was prescribed.) The capability to update past or future values of the database requires handling the following three issues:

Simultaneous Values: a situation of *simultaneous values* of a single data item in a temporal database occurs when multiple values that are valid at the same (real-world) time were assigned to a data item at different times over the database history. *Simultaneous values* is a temporal notion that may exist implicitly in non-temporal databases, with a fixed and implicit semantics to handle such a case. Temporal databases allow to refine the semantics associated with simultaneous values, which requires a supporting update model. The concept of SVS (Simultaneous Values Semantics) defined in this paper supports the different possible answers to the question: *which of the simultaneous values should be returned, as a response to a retrieve operation?* A single abstraction that captures the possible answers to that question, coupled with linguistic features to define and modify them are discussed in this paper.

Modification Control: In certain cases, it is required to restrict the ability to update the past and future states of the database. This can be done both in a static way (for all the instances of some data item in the database) or in a dynamic way (both at the object level and at the property level). Linguistic abstractions for modification control are also discussed.

Revision Control: In temporal databases, any information that has ever been stored in the database is kept and not deleted when newer information is available for the same data item (deletions are likely to eventually occur, but they are considered storage management operations). However, erroneous values, that have been corrected later, are also kept. A concept of **revision** distinct from the concept of **modification** should be established in order to retrieve the appropriate values relative to an observation time that is later than the revision time.

In this paper we devise a model that help the system designer and the users of such systems to cope with these complexities by providing enhanced schema language and enhanced set of update operation types.

This section serves as an introduction and motivation, and consists of the following subsections. Section 1.1 provides background and basic definitions in the context of temporal databases, Section 1.2 presents a motivating example, Section 1.3 outlines the rest of the paper.

[1] We use the term *data item* to denote a basic, not necessarily atomic, unit stored in the database, regardless of the specific data model. Examples: field, attribute, column.

1.1 Background: Temporal Databases

Time plays an important role in various application areas including decision support, decision analysis, computer integrated manufacturing, computer aided design, office information systems, to name a few. Due to its complexity, the functionality required by many of those applications is only partially supported by current database technology, resulting in the use of ad-hoc and expensive solutions for each application.

The temporal database research area draws an ever growing attention in the research community, summarized in a series of detailed bibliographies [Kli93], [Soo91] and [TK96]) and a survey ([OS95]). Taking an historic perspective, the incorporation of time in databases began in the '70s [FD71]; [Bra78]; [D$^+$79]. Research has concentrated on extending the relational model to include the time concept during the '80s; a survey of algebras is introduced in [MS91]. The modeling of temporal databases has been addressed in many papers including [KL83], [CC87], [Gad88], [SS88], and [CK94]. The reader is referred to the book [TCG$^+$93], for some basic readings on the subject.

In June 1993, the temporal database community organized the "ARPA/NSF International Workshop on an Infrastructure for Temporal Databases", which was held in Arlington, Texas. The results of the workshop were documented in an infrastructure recommendations report [PSE$^+$94], and a consensus glossary of temporal database concepts [J$^+$94]. Since then, substantial efforts have been invested in improving and unifying the many existing temporal query languages (e.g. HSQL [Sar93] and TQUEL [Sno87]) through a unified temporal relational query language called TSQL2[S$^+$94]. The infrastructure work forms a basis for temporal technology development, such that additional functionality can be incorporated either through mapping to or augmentation of such infrastructure. Another workshop was held in Zurich in September 1995 for further discussion of these issues[SJS95].

Following these works, we adopt a discrete model of time [CT85], which is isomorphic to the natural numbers. In this paper we use the following terms:

Definition 1.1 Chronon *[J$^+$94] is a nondecomposable unit of time, whose granularity is application dependent.*

In our case study we use the composition of *date;hh:mm* (date, hour and minute) to designate a chronon; *date* is specified as: *MMM DD YYYY*, where: *MMM* designates a month's abbreviation (e.g., Feb), *DD* designates the day in the month, and *YYYY* designates the year.

Definition 1.2 time interval *designated as $[t_s, t_e)$ is a set of all chronons t such that $t_s \leq t < t_e$.*

The temporal infrastructure advocated a bi-temporal database model, in which each data item associated with two temporal dimensions, **valid time** and **transaction time**.

Definition 1.3 Valid Time (t_v) *designates the collection of chronons at which the data item is considered to be valid in the modeled reality.*

Definition 1.4 Transaction Time (t_x) *designates the time when a data item is stored in the database. The transaction time is a single chronon designating the commit time of the transaction that inserted the data item's value into the database.*

The valid time is expressed using a *temporal element* [Gad88], which is a finite union of mutually disjoint time intervals. The transaction time is a chronon.

1.2 A Motivating Example

We start our discussion by introducing a motivating example that demonstrates the required functionality.

The case study is a decision analysis system designed to support the analysis of the decisions of physicians while treating patients in an emergency room. A patient is admitted to the emergency room. A medical record is generated and assigned to a physician. A physician determines a diagnosis that include possible alternative disorders, based on the signs and the symptoms of the patient, and on laboratory tests. In many cases, although a patient suffers from a single active disorder, the physicians must consider several possible disorders during the process of treating such a patient. The reasons for that are incomplete information, the limitations of modern medical knowledge and the lack of resources for more thorough analysis. The treatment administered to the patient should be consistent with all the possible disorders that were identified, and should not damage the patient by causing further physical disturbances. The treatment must deal at least with one or several most probable disorders. Figure 1 shows an object-based partial schema of the case study, the underlined properties designate identifiers (foreign keys).

To simplify the presentation, the properties' types are omitted. The class *Medical-Record* represents the knowledge about a single admission of a patient to the emergency room. It consists of *Symptoms* reported by the patient, *Signs* that were found in a preliminary examination by the physician and *Laboratory-Tests*. A *Diagnosis* is composed of a set of alternative *Disorders*. The *Assigned Physician* is responsible for the diagnosis and treatment. Medical records are frequently re-assigned to other physicians before the patient leaves the emergency room. The class *Patient* represents general details about patients. The class *Assigned Physician* represents the history of assigned records to a physician. The schema is presented in an object-based notation, *Laboratory-Test* and *Diagnosis* are compound properties composed of other properties.

The main goal of this application is to check if decisions made by physicians are consistent with the knowledge that was available at the time that the decision was made. This decision analysis is important for training purposes, medical research, and investigation of medical malpractice. The required functionality is demonstrated in the following investigation case:

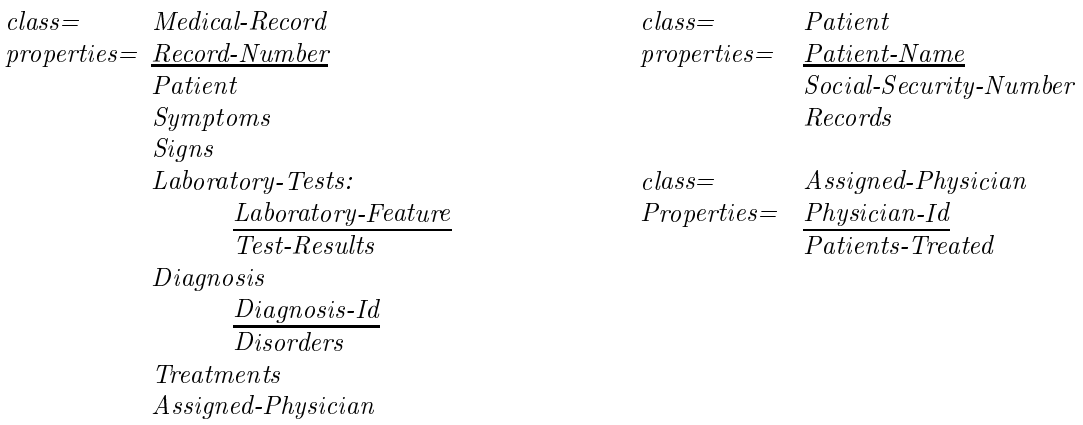

Figure 1: A partial schema of a medical database

A patient named Dan Cohen arrives at the emergency room at 10:00pm, and is assigned to Dr. Livingston. Based on the preliminary examination the physician made diagnosis D_α at 10:05pm that consisted of three possible disorders. Based on this diagnosis the physician prescribed treatment to the patient and ordered some laboratory tests. Based on these lab results, Dr. Livingston made the diagnosis D_β at 11:15pm that consisted of three possible disorders, distinct from those diagnosed in D_α.

Based on this diagnosis, a different treatment started at 11:17 pm. The diagnoses are manually reported to the data entry center, whose responsibility is to enter each report into the database. In the data entry center, reports are accumulated and are reported to the database in batches. Due to a shift change around 11:00 pm, there was a delay of reports of some diagnoses including D_α, that was committed in the database only at 11:35 pm. The diagnosis D_β was committed without outstanding delay at 11:30 pm. In order to investigate the physician's actions the following functionality is required:

1. To answer a query such as:

   what was the valid diagnosis when a treatment was administered?

2. The ability to determine the sequence of events that occurred in the modeled reality, even if it is distinct from the sequence of reports of these events to the database. For example, we should be able to trace the fact that the D_α disorder was diagnosed before the D_β disorder and that certain treatment was given when D_α was diagnosed, and D_β had not yet been diagnosed.

1.3 The Paper's structure

Figure 2 shows an EER(Extended Entity Relationship) diagram of the components described in this paper. The three requirements specified in this section are defined in Section 2. These requirements should be satisfied both by static definitions, at the schema level, and by dynamic definitions that are materialized in run-time update operation types. Section 3 describes the information model primitives. representation and linguistic aspects. Section 4 discuss in detail the retrieval and update semantics including extended set of update operation types. Section 5 discusses implementation issues. Section 6 concludes the paper and discusses the model vs. its requirements.

2 The Required Extended Update Functionalities

This Section discusses the three required extended update functionalities. Section 2.1. discusses the concept of simultaneous values, section 2.2. discusses the modification control issue, Section 2.3. discusses the revision issue,

2.1 Simultaneous Values

In the motivating example, there are several properties that require different interpretations of handling simultaneous values. We start our discussion with the definition of this term.

Definition 2.1 Simultaneous values of δ at t: *A data item δ is said to have simultaneous values at a chronon t, on the valid time dimension, if n update operations were issued over the database history assigning to δ the values $\{v_1, \ldots, v_n\}(n > 1)$, such that $\forall i : t \in t_v(v_i)$.*

Note that the n updates are a subset of the total number of updates to that data item, and n must be at least two for simultaneous values to exist.

In Section 2.1.1. we define the term SVS, in Section 2.1.2. we introduce single value interpretations of SVS, in Section 2.1.3. we introduce multiple value interpretations of SVS. This discussion entails a need to consider an additional time dimension called **Decision Time**. This issue is discussed in Section 2.1.4.

2.1.1 Simultaneous Value Semantics

Definition 2.2 Applicable value(s) of a data item δ at t: *The value, or values that are being selected in response to a retrieval request about the value of δ at the valid time chronon t.*

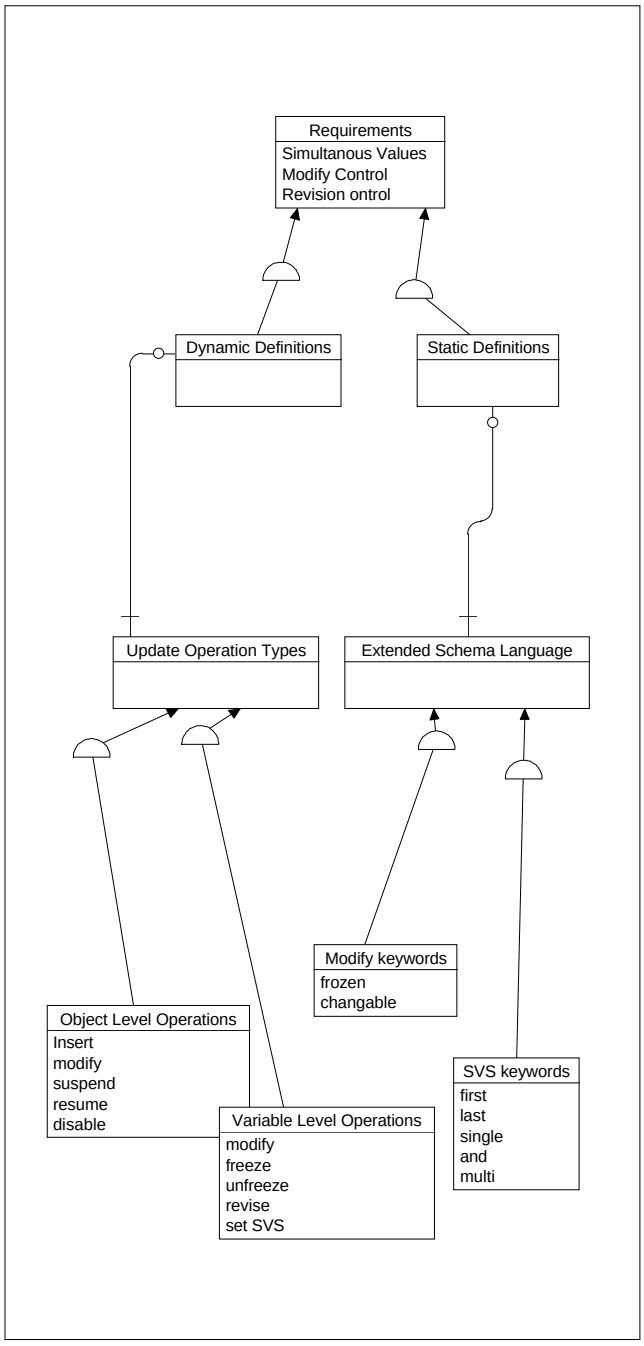

Figure 2: The model's components

Definition 2.3 SVS: *an abstraction that denotes a decision procedure at retrieval time to determine which of the simultaneous values are applicable.*

The possible values of SVS are determined according to the result of two separate decisions:

- *Should the applicable value be a single value for a given chronon t , or are multiple values allowed to be applicable?*
- *What determines which value is applicable?*

2.1.2 Single Applicable Values

If there are multiple values for a chronon t on the valid time dimension, a criterion has to be devised of how to choose a single applicable value. We discuss three possible strategies: the **first value**, **last value** and **single user defined value**. The approach that selects value on the basis of chronological order is rooted in the semantics of the **insert** and **modify** operation types in conventional databases.

Definition 2.4 first value semantics: *The first known value of the data-element δ at the chronon t is the only applicable value.*

The term *first known value* may be implemented using the earliest value on the transaction time (t_x) dimension, however this interpretation designates the order in which the values have been reported to the database. This order may be different from the real order of events. Applications for which this distinction is significant, should employ another time dimension to maintain the correct order. This issue is further discussed in Section 2.1.4.

The **insert** operation type in conventional databases is a case of **first value** semantics. In conventional databases, the first **insert** operation for a given primary key value is stored in the database, while later **insert** operations of the same instance are considered as integrity constraint violations and thus rejected. At the attribute level, **first value** semantics exists in unchangeable attributes [N+87].

In temporal databases, a kind of **first value** semantics update protocol as proposed in [Sno87] is illustrated in Figure 3. The two bottom lines represent update operations, and the upper one represents a response to a retrieve operation. Note that the values denoted by + and * have not been overridden by the later update, due to the **first value semantics**.

Definition 2.5 last value semantics: *The last known value of the data-element δ at the chronon t is the only applicable value.*

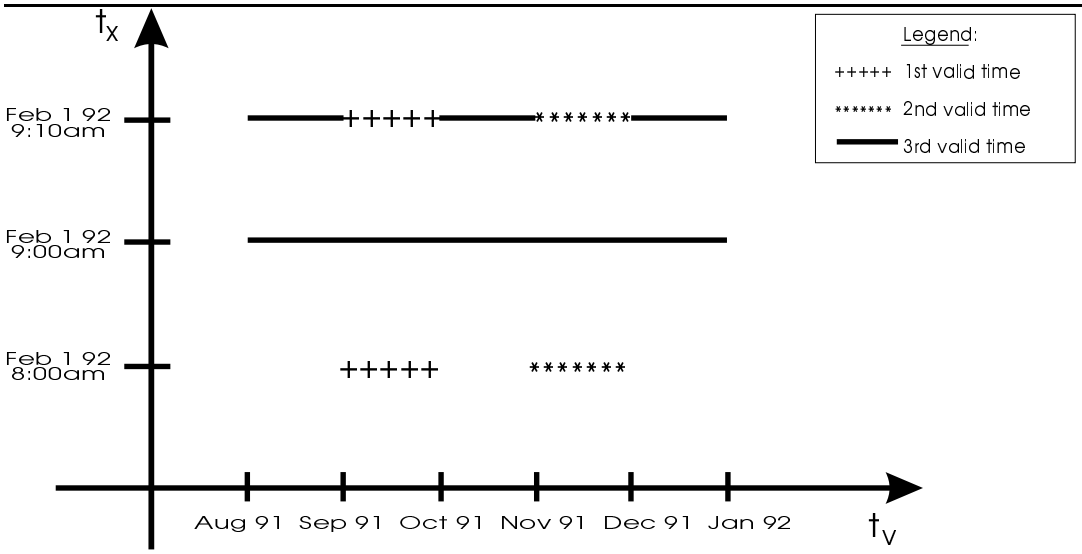

Figure 3: A fixed first value semantics update protocol

The **last value** semantics is compatible with the **modify** operation type in conventional databases, the last modified value overrides all the previous ones.

Some temporal models (e.g, [SK86], [WJL91]) employ **last value** semantics, in which the value with the latest transaction time is considered to be the applicable value.

Definition 2.6 single user defined value semantics: *The selection criterion among the values is provided by the user using a query language. The selection is constrained to any selection criterion that selects a single value (which can be an aggregate value).*

Examples: the minimal value, the median value, the average value, a value that is selected according to other parameters, such as source and level of confidence, if such information is stored in the database [GES94], and an explicit selection by the user. A special case is to look at the different values as possible interpretations, in which only one is applicable. This is compatible with the *possible worlds* semantics that has been thoroughly discussed in the knowledge representation area [F+94], and mentioned also in the database context [AKG91]. Retrieval queries, under this interpretation, can include modal operators.

2.1.3 Multiple Valued Approach

In the multiple valued approach, there is no restriction on the number of applicable values. Any subset of the set of simultaneous values of the data item δ at chronon t may be selected. There are two types of SVS applicable to this case: the **all** semantics, and the **multivalued user defined semantics**

Definition 2.7 The all values semantics: *all the simultaneous values of the data-element δ at chronon t are applicable.*

Under the **all values semantics**, all the values (except for the revised ones) are being selected. This approach was referred to in semantic data models as a *multi-valued* attribute, in which a data item's value consists of several values [HK87]. For example, a data item that designates the languages that a person speaks can have a set of grouped values. The interpretation is **All**, designating that the person speaks all the languages in the designated set.

Definition 2.8 Multiple user defined value semantics: *The selection criterion among the values is provided by the user using a query language.*

Examples to selection criteria are: all values > 5, all values whose transaction time is earlier than t_0, and the differences between the original values and their average. Similar notion was defined in databases that support uncertain multiple values [ZP93]. In our case study, the physician's diagnosis consists of a set of disorders, any subset of which may reflect the patient situation. In this case, the values employed as part of the retrieval requests by some selection criteria or aggregation of values.

Current temporal database models employ a single interpretation of simultaneous values either at the database or at the schema level. Some of the models (e.g., [Sno87],[NA89]) enforce a **single value approach** at the database level. Other models (e.g., [Tan86]) enable the support of different semantics at the schema level by making a distinction between a single valued attributes and a multi-valued attributes; however, the semantics of the multi-valued attributes is not explicit. Many other models (such as [Ari86], [SK86], [ABN87], [WJL91], and [CK94]) also enforce a single value for each chronon. As a result, a mechanism to handle non-unique interpretations is not available. In [CK94], a mechanism for storing multiple past views is provided, but a predefined preference relation for choosing a single value of a property for each chronon is enforced. Our case study demonstrate the need for all the spectrum of simultaneous values semantics.

- The *Physician-id* follows the **first value** semantics;
- The *Symptoms* property has an **all values** semantics; all the reported symptoms are considered to be applicable.
- The *Diagnosis* has a **last value semantics**. The last diagnosis is the applicable one. However, as shown in our example, the transaction time is not necessarily a good measure in determining the correct order of diagnoses.
- The *Disorder* property within a context of a single *Diagnosis* has a **multivalued user defined semantics**; any subset of the set of disorders may be applicable.

- The *Patients-Treated* for a physician has a **single user-defined semantics** with respect to the question: which patient is being treated by Dr. Livingston at chronon t. If we assume that a physician can handle a single case in any given chronon, then it is known that at chronon t he treated **one** of the patients whose assignment to him is valid at t.

These examples show cases of SVS that can be determined during the schema design phase. However, in some cases the semantics should be determined only at run-time. For example, Dr. Flinstone is not allowed, from a certain date on, to be responsible for more than one patient, thus for this instance, the assignment semantics should be modified to **first value semantics.**

As a requirement, the model should provide both static and dynamic SVS definition. The static definitions are implemented at the schema level (with a possible schema evolution), and the dynamic level is implemented as updates at the instance level.

2.1.4 Decision time

Some of the **SVS** options are based on the order of events, which lead us to discuss another important aspect for the required functionality, the issue of decision time.

The transaction time (t_x) in some temporal database models has two major roles (in addition to the traditional role of backup and recovery):

- It is used to determine the order of events, necessary to support **first value** or **last value** semantics;
- It is used to answer temporal retrieval queries, such as:

  What was the answer to the query q, if issued from the observation point of a past chronon t?

  To answer such a temporal query, the database is required to know the values committed before t; the transaction time is a means to record this knowledge.

The second role concerns events in the database domain only (the commit time), thus the transaction time can be used for achieving this role without additional assumptions. However, the first role may refer to events in the application domain and not to the database domain. There is an implicit assumption that the transaction time reflects the correct order of events in the application domain. Thus, the transaction time is sufficient to achieve the first role. Contrary to that, there are applications in which the order of events is important and the order of updates to the database does not necessarily reflect the order of events in reality; in this case, we need a time type that belongs to the application domain and not

to the database domain. In our case study example, diagnosis D_α occurred before diagnosis D_β, but due to the batch process of reporting, diagnosis D_β was committed in the database before diagnosis D_α. In general, the commit order of transactions is non-deterministic under the standard two phase locking protocol, consequently the transaction time may not reflect the order of occurrences in the modeled reality. In our context, the decision analysis context, we use the term **decision time** for this time type.

Definition 2.9 Decision Time (t_d) *is the chronon at which a data item's value was decided in the application's domain of discourse [EGS92].*

This chronon denotes the time at which an event occurred, or the time at which a decision was made (even if the value is complex, a decision about each modification is made in a single chronon). From the database's point of view, t_d reflects the chronon at which an event in the modeled reality entails a decision to initiate a database update transaction.

The following example shows the three different types of times. Dr. Flinstone is hired as a physician in our hospital, the hiring decision has occurred on July 20 1996, and recorded in the database on July 24 1996. The hiring period is for a year starting August 1 1996. In this case t_d = July 20 1996, t_x = July 24 1996, and t_v = [August 1 1996, July 31 1997).

We assume that the **decision time** dimension is the one, according to which the **first** and **last** value semantics is being determined, this means that the value having the earliest (or latest) **decision time** is the applicable one.

The decision time concept was introduced in [EGS92], and is also mentioned in [OS95]. A similar concept has been referred to as *event time* [CK94]. It is argued in [OS95] that it is still an open question whether the functionality achieved by using decision time, as a third time type is justified with respect to its overhead. In this paper we assume that the system designer and user recognize the decision time as a primitive concept; the discussion about implementation as a separate concept vs. implementation on top of existing concepts is deferred to Section 5.

2.2 Modification Control

In temporal databases, values that are valid in the past or the future may be updated. While this ability provides flexibility, it is sometimes required to restrict it and not allow to modify data items during part or all of their validity time. For example, actions that have been performed, such as the values of laboratory tests that have been reported, cannot be altered. Thus, this data item is unchangeable in the entire valid time dimension. In other cases, a data item can be changeable in some valid times and unchangeable in other valid times. The modification control can be issued either in a static way or in a dynamic way, and either at the object level or the property level.

Definition 2.10 *A* **static modification control** *is a modification control that applies to all instances of the class (or property) of the same type for any chronon on the valid time dimension.*

Definition 2.11 *A* **dynamic modification control** *is a modification control that overrides the* **static modification control** *for a certain object (or a data-item) during some chronons on the valid time dimension.*

In a similar way to the SVS case, The **static modification control** is implemented by a modified schema definition (with a possibility of evolving schema) and the dynamic one is implemented at the instance level.

2.3 Revision Control

The revision requirement is a result of the ability to ask queries from different view points, example for such a query is: what are the known symptoms of the patient John Galt, as was known at Dec 12, 1995; 10:00 pm. Such a query is vital for decision analysis and auditing purposes. In regular database, the last value overrides the previous one, thus it is not important whether the value was replaced because some change had occurred, or the value was replaced because it was erroneous. However, in temporal databases this distinction is important. Consider the following example: The symptoms S_a, S_b, S_c for a certain patient were reported to the database at the chronon t_1. at $t_2 > t_1$, it was noticed that the symptom S_b had been reported by mistake, and it should have been reported as S_d. The requirement is that a query issued from the observation point of any chronon t, such that $t_1 < t < t_2$, about this patient's symptoms should return $< S_a, S_b, S_c >$, while the same query issued from the observation point of $t \geq t_2$ should return $< S_a, S_d, S_c >$. This is consistent with the knowledge that can be obtained from the database at each observation point. In our example, the value was replaced with another value for its entire validity time, but in the general case the **revision control** should allow either revision by another value, or just logical deletion of the revised value. The revision may apply to the entire validity time of the revised value, or to any part of it. The **revision control** is implemented at the instance level, dynamically.

3 The Modeling Primitives

In this section we present the primitives of the temporal database model that is intended to satisfy the requirements posed in the previous sections. These primitives are used by the system designer when constructing the application. This issue is further elaborated in Section 4. Section 3.1 presents the information modeling primitives. Section 3.2 discusses the enhanced schema language support for the static SVS and modification control definitions,

Section 3.3 introduces the set of update operation types, which are the major implementation vehicle for the dynamic SVS and modification control definitions. The semantics of these components is discussed in Section 4.

3.1 Information Modeling Primitives

This section presents the information modeling primitives that are used in this paper. This data model can be implemented on top of various lower-level data models, such as relational or object-based.

Information about an object is maintained as a set of variables (instances of the class' properties). Each variable contains an information about the history of values as well as the different components of the variable status (SVS, modification control, revision control) of the variable. Each component is represented using a set of state-elements; state-element is the most basic object in the database.

We assume that the database is an *append only* database. New information is added while existing information is left intact. The *append only* approach is necessary to support operations that require past database states. For example, a medical examiner investigating a malpractice complaint issues the query:

"What were the known laboratory test results of a given patient at 10:30pm on December 12, 1993?"

This information is crucial in deciding whether the attending physician provided a reasonable treatment **given the available information at that time**. Since the information may have been incomplete or even erroneous at the time, the treatment decision may seem wrong from a later observation point. Unlike some other temporal models [ABN87] that employ a non-strict form of append-only, we employ the append-only in the strictest fashion. Consequently, the data can be stored on WORM (write once read many) devices, in which no changes can be made to a state-element after the transaction that created it had committed.

A state-element is a tuple of the form:[2]

$$\langle \textit{se-id, oid, value, } t_x, \, t_d, \, t_v \, \rangle$$

- t_x, t_d, t_v designate the time types (as defined, t_x and t_d are chronons and t_v is a temporal element).

[2]Additional attributes of information about source, validity, accessibility, etc., can be added. These extensions are discussed in [GES94].

- The *value* of a state-element designate a value assigned to the variable (e.g., Dr. Livingston),
- A state-element includes a uniquely created system-wide identifier *se-id*.
- *oid* designates the object-identity of the object the state-element is associated with.

A state-element example is:

Treatment=
se-id=s9, oid=864545, value=antibiotic,
t_x=Dec 12 1993; 10:30pm, t_d=Dec 12 1993; 10:10pm, t_v=[Dec 12 1993; 10:12pm, Dec 19 1993; 8:00pm)

A Bucket β is a set of state-elements having a well-defined semantics. In our model there are four types of buckets, as defined below.

A variable δ is as a set of four buckets:

$$\langle \delta.data,\ \delta.variable\text{-}SVS,\ \delta.modify\text{-}control,\ \delta.void\text{-}SE \rangle$$

The *data* bucket contains the state-elements whose values issue the history of the data associated with the variable δ. The rest of the buckets are control buckets. The variable-SVS contains state-elements whose value designate dynamic modifications of the SVS of the variable δ. The values consists of a pair ⟨SVS, query-id ⟩. The query-id designates a query to be activated for user defined SVS. The *modify-control* is a collection of state-elements whose value (changeable or frozen) designate the history of modifications to the variable's modify control status. The *void-SE* is a collection of state-elements, whose value are state-elements that are being voided at the t_v of the void state-element.

An object α is represented as a set of variables:

$$\langle \alpha.object\text{-}id,\ \alpha.class\text{-}ref,\ \alpha.object\text{-}status,\ (\alpha.p_1, \ldots, \alpha.p_n) \rangle.$$

The data bucket of the *object-id* variable consists of a single unique state-element whose value designates the object identity. Its modify-control bucket consists of a single state-element with the value frozen. The *class-ref* is a variable that classifies an object to be an instance of a specific class. The SVS of this variable can be adjusted to the specific application's assumption. If an object can be classified to multiple classes, then the SVS of *class-ref* is set to **AND**; if an object's classification is fixed then the SVS is set to **first value** SVS. This is an example of using the SVS concept to support data model independence. The *object-status* variable's values are stored in state-elements, with **last value** SVS, based on **decision time**. The possible values of this variables' data are: **active, suspended, disabled**. See Section 4 for the exact definition.

An object's state is a set of all its variables' states, i.e the entire collection of state-elements associated with this object. In the general case, the user may not be familiar with the object-identity, and instead identifies the object using an *object identifier* (primary key), which is a subset of the object's state. For example, the underlined properties (*Record-Number* and *Patient-Name*) in Figure 1, are the object-identifiers.

The level of granularity of temporal support was discussed in various papers (e.g. [SA86]). The common claim is that an attribute level support (which is equivalent to our interpretation of a state-element) reduces the space complexity relative to an object level support, because any change in any attribute results in the need to duplicate the entire object, also if the level of granularity required in the application is of an attribute, then an object level support increases the time complexity of obtaining information about the evolution of a single attribute. In any event, the concepts discussed in this paper are model independent, the concept of **state-element** can also be implemented on top of a model whose temporal granularity is in the object level, by creating an object to represent each state-element.

3.2 The Enhanced Schema Language

The schema language is the system designer's tool to express static decisions about the data representation and semantics of updates and retrieval requests. The schema definition consists of classes and properties; each property may have characteristics that are common in existing schema languages (e.g., type, default, set of legal values, reference to other objects), and additional characteristics required to support the static definitions of extended requirements (SVS and modification control) By using keywords.

The SVS keywords are: **first, last, and, single, multi**. The **single** and **multi** keywords designate the user defined SVS modes. An additional keyword **query = qid** is allowed with the **single** and **multi** SVS options, to designate the id of a query that is activated,[3] whenever a query is issued that require the value of any variable that belongs to this property. *qid* is a query id. Example: if a property p has a **single** SVS mode associated with it, and the query associated with it is average [value], then anytime that any query attempts to retrieve any instance of p, the average of the values of all the state-elements valid at the specified valid time are returned. If none of the SVS keywords is specified then the default is **last**. If a **single** or **multi** SVS have been specified, and no query has been indicated, then the user is prompted at run-time for a selection query[GES94].

The **modification control** employs two keywords: **frozen** and **changeable**. The default is **changeable**. In Figure 4, we re-visit the schema presented in Figure 1 with the additional keywords. Since **changeable** is the default, it is omitted. Note that a nested structure can have a different SVS in the different levels; *Diagnosis* obeys the **last value** SVS, while its component *Disorder* has a **multi** SVS, consequently there can be only a single valid *Diagnosis* at each single chronon, nevertheless, within this *Diagnosis* multiple disorders

[3] queries are represented as objects in the database.

class= *Medical-Record*
properties= *Record-Number:* **last**
Patient: **first; frozen**
Symptoms: **all**
Signs: **all**
Laboratory-Tests: **all**
Laboratory-Feature **first; frozen**
Test-Results: **all; frozen**
Diagnosis: **last**
Diagnosis-Id: **first; frozen**
Disorders: **multi**
Treatments: **last**
Assigned-Physician: **last**

class = *Patient*
properties = *Patient-Name:* **last**
Social-Security-Number: **last**
Records: **all**

Class = *Assigned-Physician*
properties = *Physician-Id:* **first; frozen**
Patients-Treated: **single**

Figure 4: The revised partial schema of a medical database

may be simultaneously valid.

In this example all the properties SVS were explicitly defined. To ease the system designer task, we suggest to use the following defaults that are compatible with update assumptions in conventional databases:

1. When the property is an object-id, the default is **first; frozen value** (this is an unchangeable default).
2. When the property is an object-status (see Section 3.3), the SVS is **last value; changeable** (this is an unchangeable default).
3. If the data type of the property is a set, a bag or a sequence, then the default is **all; changeable**. In this case *insert* means add a new element, while *modify* means change existing element(s).
4. If the data type of the property is an atomic data type, then the SVS is **last; changeable**.

The extended schema language supports static definitions of the required options. These definitions affect all instances of the properties defined in the schema, unless a dynamic definition overrides it. The schema level is not entirely static, in the sense that a schema may evolve with time, although we assume that schema changes are not frequent. If a schema evolves, the valid schema is used. For a comprehensive discussion of the schema evolution issue the reader is referred to [GE98].

3.3 The Update Operation Types

Update operation types are the linguistic primitives of a database update language. We express the required dynamic functionality by augmenting this set of primitives, hence, providing the user a uniform linguistic commands for the entire update process that include update of data, modification control at the object and variable levels, revision control and SVS definitions.

Earlier works in the temporal database area were confined to the update operation types of *insert, modify* and *delete* while assigning to these operations a slightly different meaning than in conventional databases. For example, in several works (e.g., [EW90]) the difference between updates in non-temporal databases and in temporal databases is that modifications of an attribute's value in the latter case retain the old value in addition to adding the new value. Others (e.g., HRDM [CC87], [McK88], [GE98] expanded the *modify* operation to include meta-data, thus allowing schema versioning, as well as data evolution. Our extended set includes the **insert, modify, suspend, resume, disable, freeze, unfreeze, revise, set-SVS** operations, as explained next.

Insert: This operation creates a new object in the database. Along with the object insertion, the user may assign initial data values to some or all the object variables. For example, a new patient is registered at the emergency room. The database creates a new instance of the class *Patient* and initializes the values *Patient-Name=Dan Cohen* and *Social-Security-Number=12345678*.

Modify: This operation adds new information about an existing object. For example, in Dec 12, 1993, 11:10pm, the results of a laboratory test of Dan Cohen caused a modification to the *Diagnosis* variable. Unlike non-temporal databases, the **modify** operation does not remove previous values. The modify operation can be applied to valid time chronons that are different than *now*, to an interval, or even to the entire database valid time line.

Suspend: This operation establishes a reversible constraint that prevents any modification to the object in the given valid time, except for the object status which is still changeable.[4] For example, we can use the **suspend** operation to prevent the assignment of a treatment until the completion of appropriate tests. The **suspend** operation is a modify-control operation that sets an object to be **unchangeable** For example, when a physician is off-duty it is not possible to assign any record to him.

Resume: This operation makes a suspended object changeable again. As in the *insert* operation, the *resume* operation may be used to set the values of some of the object's variables. The **resume** operation is necessary to eliminate an **unchangeable** constraint of an object.

Disable: An operation that establishes an irreversible constraint that makes the object logically deleted as of the beginning of the t_v specified in the disable operation, and consequently prevents any modification to the specified object. For example, when a physician retires (assuming that a retired physician cannot practice again), the object representing this physician is disabled, however we may still want to investigate his past action, thus the history of records assigned to him is kept. The **disable** operation type has two major differences from the **suspend** operation type:

- **disable** is irreversible;[5]
- **disable** has ontological implications, because it means that an object is logically deleted, i.e. ceases to belong to the application's domain of discourse, while *suspend* is only a constraint that prevents updates.

We use the term *disable* rather than *delete* since the history of the disabled object is preserved and there are no physical deletions.

Freeze: This operation establishes a reversible constraint that prevents the modification of a variable (except in the case of revising erroneous values as explained below).[6]

[4]The object status is required to remain changeable in order to reverse the suspend constraint.

[5]A Database Administrator (DBA) can use low level update primitives to "rescue" an object that was mistakenly disabled.

[6]The **freeze** and **unfreeze** operation at the variable level are similar to the **suspend** and **resume** at the object level. The different names are intended to avoid semantic overloading.

For example, the laboratory results are measured values that should not be altered, thus the laboratory results' variable is updated with a *freeze* constraint. The **freeze** operation is vital to the support of the **unchangeable value** at the variable level.

Unfreeze: Any frozen data may be unfrozen. An *unfreeze* operation applied to a variable, designates the removal of the freezing constraint. Any modification to that variable is allowed from that time on. The **unfreeze** operation is required for the retraction of the **unchangeable value** constraint at the variable level.

Revise: This operation "corrects" an erroneous value of a variable at certain collection of chronons. It tags values that currently exist in the database as false ones and adds a new correct value instead. The revise operation allows the replacement of a frozen value, marking the previous value as an erroneous one. The *revise* operation type is the means to implement the revision control requirement. The separation of the *revise* operation from the *modify* operation makes a semantic distinction between a change in the real world and a correction of a wrong value that was reported to the database. The user can instruct the database to include or exclude the revised values in retrieval operations.

Set-SVS: The operation dynamically sets an SVS at the variable level.

Data may only be changed in a temporal database by adding new objects or adding new state-elements to the variables of an existing object. The semantics of the update model are reflected in allowable new state-elements. A new state-element is allowed to be inserted if it obeys some general syntactic rules, such as legal value in its valid time, and other rules that are contingent on the status of the object and the variable, the update operation type, and the SVS for this variable. Section 4 discusses the exact semantics of each update operation.

4 The Semantics of the Model's Components

In this section, the formal update semantics of the various components of the model is presented. The validity semantics is presented in Section 4.1, the retrieval semantics is presented in Section 4.2, the update operation types are combined from a set of low-level primitives, presented in Section 4.3. Section 4.4 describes the semantics of the update operation types, followed by a discussion in Section 4.5.

We shall use Figure 5 to demonstrate each of the functions and operations, presented in this section. The figure presents a set of state-elements, labeled according to the *se-id,* of an object that is an instance of the *Patient* class. The *se-id* are identified as s_{nn}. Each state-element is preceded by the name of the bucket it belongs to.

Object-id.data
(s1) 864545, t_x=Dec 12 1993; 10:02pm, t_d=Dec 12 1993; 10:00pm, t_v=[Dec 12 1993; 10:00pm, ∞)
Class-ref.data
(s2) Patient, t_x=Dec 12 1993; 10:02pm, t_d=Dec 12 1993; 10:00pm, t_v=[Dec 12 1993; 10:00pm, ∞)
Object-status.data
(s3) Active, t_x=Dec 12 1993; 10:02pm, t_d=Dec 12 1993; 10:00pm, t_v=[Dec 12 1993; 10:00pm, ∞)
Patient-Name.data
(s4) Dan Cohen, t_x=Dec 12 1993; 10:02pm, t_d=Dec 12 1993; 10:00pm, t_v=[Dec 12 1993; 10:00pm, ∞)
Social-Security-Number.data
(s5) 12345678, t_x=Dec 12 1993; 10:02pm, t_d=Dec 12 1993; 10:00pm, t_v=[Dec 12 1993; 10:00pm, ∞)
Social-Security-Number.Modify-Control
(s6) frozen, t_x=Dec 12 1993; 10:02pm, t_d=Dec 12 1993; 10:00pm, t_v=[Dec 12 1993; 10:00pm, ∞)
Record-Number.data
(s7) 12345678-1, t_x=Dec 12 1993; 10:02pm, t_d=Dec 12 1993; 10:00pm, t_v=[Dec 12 1993; 10:00pm, ∞)
Record-Number.Modify-Control
(s8) frozen t_x=Dec 12 1993; 10:02pm, t_d=Dec 12 1993; 10:00pm, t_v=[Dec 12 1993; 10:00pm, ∞)
Treatment.data
(s9) antibiotic, t_x=Dec 12 1993; 10:30pm, t_d=Dec 12 1993; 10:10pm, t_v=[Dec 12 1993; 10:12pm, Dec 19 1993; 8:00pm)
Disorder.data
(s10) partial treatment,
t_x=Dec 12 1993; 11:30pm, t_d=Dec 12 1993; 11:15pm, t_v=[Dec 12 1993; 11:15pm, ∞)
Disorder.data
(s11) brain abscess, t_x=Dec 12 1993; 11:30pm, t_d=Dec 12 1993; 11:15pm, t_v=[Dec 12 1993; 11:15pm, ∞)
Disorder.data
(s12) viral Meningitis, t_x=Dec 12 1993; 11:30pm, t_d=Dec 12 1993; 11:15pm, t_v=[Dec 12 1993; 11:15pm, ∞)
Social-Security-Number.Void-Se
(s13) s5, t_x= Dec 12 1993; 11:33pm,t_d=Dec 12 1993;11:30pm, t_v=[Dec 12 1993; 11:30pm, ∞)
Social-Security-Number.data
(s14) 02345678, t_x=Dec 12 1993; 11:33pm, t_d=Dec 12 1993; 11:30pm, t_v=[Dec 12 1993; 11:30pm, ∞)
Disorder.data
(s15) bacterial Meningitis, t_x=Dec 12 1993; 11:35pm, t_d=Dec 12 1993; 10:05pm, t_v=[Dec 12 1993; 10:05pm, ∞)
Disorder.data
(s16) viral Meningitis, t_x=Dec 12 1993; 11:35pm, t_d=Dec 12 1993; 10:05pm, t_v=[Dec 12 1993; 10:05pm, ∞)
Disorder.data
(s17) spontaneous Subarachnoid Hemorrhage,
t_x=Dec 12 1993; 11:35pm, t_d=Dec 12 1993; 10:05pm, t_v=[Dec 12 1993; 10:05pm, ∞)
Treatment.data
(s18) acyclovir,
t_x=Dec 12 1993; 11:35pm, t_d=Dec 12 1993; 11:17pm, t_v=[Dec 12 1993; 11:19pm, Dec 22 1993; 8:00am]
Record-Number.modify-control
(s19) changeable, t_x=Dec 13 1993; 10:02pm, t_d=Dec 13 1993; 10:00pm, t_v=[Dec 13 1993; 10:00pm, ∞)
Object-Status.data
(s20) Suspended, t_x=Dec 19 1993; 8:02am, t_d=Dec 19 1993; 8:00am, t_v=[Dec 19 1993; 8:00am, ∞)
Object-Status.data
(s21) Active, t_x=Aug 24 1994; 12:03am, t_d=Aug 24 1994; 12:00am, t_v=[Aug 24 1994; 12:00am, ∞)
Object-Status.data
(s22) Disabled,
t_x=Aug 25 1994; 8:05am, t_d=Aug 25 1994; 8:00am, t_v=[Aug 25 1994; 8:00am, ∞)
Object-Status.modify-control
(s23) freeze,
t_x=Aug 25 1994; 8:15am, t_d=Aug 25 1994; 8:12am, t_v=[Dec 19 1993; 8:00am, ∞)

Figure 5: An example set of state-elements

4.1 Validity Semantics

An object is considered to be **active** at chronons in which it is neither disabled nor suspended on the valid time axis. The state transition diagram of the object-status is presented in Figure 6. An arrow's label represents the name of the update operation that changes the object's status. Note that the *disabled* state is a terminal state, unlike *suspended* and *active*. The variable's states are applicable only within the context of the active object status.

An object is valid when it is not disabled. When an object is disabled, all its variables are considered to be invalid, except for the *Object-Status* that continues to be valid, because it provides information about the validity of an object. In the example, the object is invalid in *[Aug 25 1994; 8:00am, ∞)*, which is the valid time of *(s22)*. A **disable** operation sets an actual upper bound for the valid time (t_v) of all the state-elements associated with the disabled objects to be the starting point of the *disabled* status valid time interval. Thus, the chronon *Aug 25 1994; 8:00am* marks the upper bound for actual valid time of all the state-elements associated with this object. Note that the recorded t_v of the state-elements cannot be modified, however, the upper bound is reflected in the update and retrieval operations semantics. An object cannot be referenced by other objects, at a valid time chronon in which it is disabled. The collection of chronons in which an object α is active or valid is denoted by AR (α) or VR (α), designating the activity range and the validity range, respectively.

A variable has a valid value only when its associated object is valid. The CSE function (Candidate State-Elements) returns the state-elements of a given variable which are valid at chronon t, i.e. the state-elements whose valid-time contains the chronon t. All these state-elements are candidates to be applicable, depending upon the SVS semantics.

Definition 4.1 CSE(var, t) *is a function that returns the set of state-elements of the data bucket of the variable* **var** *that are possibly valid at a chronon* **t** *. A state-element se belongs to this set if it satisfies the following conditions:*

1. $t \in VR(se.oid)$ */* the object is valid at t */;*
2. $t \in t_v(se)$ */* se is valid at t */;*
3. $\neg\exists se' \mid se.se\text{-}id = se'.value \wedge t \in t_v(se') \wedge se' \in var.void\text{-}se \wedge t_x(se') > t_x(se)$
   / se is not voided at t. */;*

For example, CSE(α.*Object-Status*, Aug 24 1994; 12:00am)={s3, s20, s21}, where α is the object whose state-elements are presented in Figure 5.

The applicable state-elements among those included in the CSE set are determined according to the SVS semantics. For example: in the **all** SVS, the whole set is considered to be applicable. In the **last value** SVS, the applicable state-element is a state-element whose t_d is the latest among the CSE set. t_d may be used when the variable belongs to the application

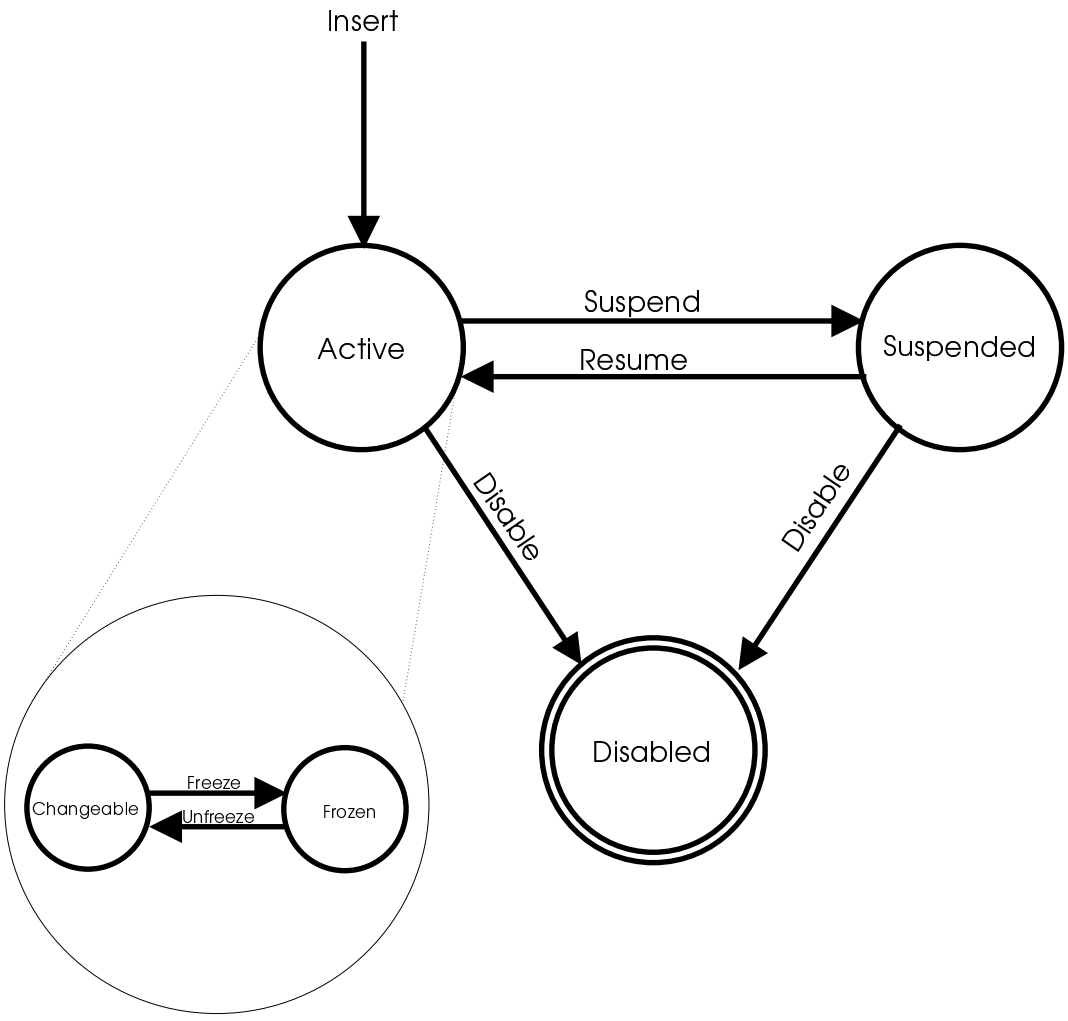

Figure 6: The state transition diagram of the object-status

domain. We denote the state-element chosen by the **last value** SVS as ASE (Applicable State Element). We assume that each decision is made at a unique chronon, thus ASE is an atom. For example, ASE(α.*Object-Status* , Aug 24 1994; 12:00am)={s21}, where α is the object whose state-elements are presented in Figure 5.

The frozen range of a variable is the range in which the variable is frozen. This is defined by the function FR (*var*). The function FR returns a collection of valid time chronons in which the applicable state-element is frozen, i.e., it cannot be altered. This function returns the unions of t_v of all sate-element in var.modify-control, whose value is "frozen".

4.2 Retrieval Semantics

The retrieval semantics is determined according to the variable's SVS, the validity semantics and additional information that may be obtained from the user. The basic retrieval request is: find the value of a variable *var* at chronon t.

By satisfying this retrieval request, many complex queries can be answered. The basic retrieval request has the following interpretation:

1. If the SVS is **first value** then the state-element with the earliest decision time among those returned by the CSE function is selected.
2. If the SVS is **last value** then the ASE function returns the value.
3. If the SVS is **all** then the set of all values in the CSE set is returned.
4. If the SVS is **user defined** then if a query is referred to at the schema or the variable level, the result of this query is returned, else the user is prompted for a selection query (in this case the SVS is deferred to run-time interpretation). An example of such query is $t_d < t_0$, which selects only the set of state-elements decided prior to t_0.

This semantics can be implemented on top of various query languages such as TOOSQL [RS91] that also support retrieval from various observation points (an answer to the query as-of t_o) that restricts the selection of values to those whose $t_x < t_o$. The following examples illustrate the retrieval semantics (all of the following queries were issued on December 13, 1993).

1. Query: What is the disorder of Dan Cohen?
   Answer: The possible Disorders of Dan Cohen are partial treatment, brain abscess, and viral Meningitis.
   The answer is based on state-elements (s10)-(s12). Since *Diagnosis* has a **last value** SVS, the diagnosis with the highest *decision time (t_d)* is selected by the ASE function. The *Disorders* within a *Diagnosis* have a **user defined** SVS, thus the answer is interpreted as possible disorders.

2. Query: What was the known Social-Security-Number of Dan Cohen at 10:30pm on December 12, 1993?
   Answer: The known Social-Security-Number of Dan Cohen on December 12, 1993 at 10:30pm is 12345678. An intelligent query language can point out that the value was erroneous, and was revised to 02345678 on December 12 1993, at 11:33pm.

4.3 Low-level Update Primitives

This section presents the low-level primitives the system use to update the database. These primitives are the building constructs of the update operation types and are not accessible to the user. However, the DBA may use these primitives in handling exceptional situations. The primitives are defined at three different levels: state-element primitives, variable primitives and object primitives. Throughout this section, we use the symbols $\oplus$ and $\otimes$. The symbol $\oplus$ denotes an application of an update operation to a database. The symbol $\otimes$ is a separator between two successive operations; in case of an abort as part of one of the operations, subsequent operations are not performed. We also use two constants, *now* designates the chronon at which an operation is being performed, ∞ designates an unlimited known upper bound, for example a state-element having a valid-time interval of $[now, \infty]$ is considered to be valid starting from the time it was inserted, and valid at any later chronon, unless voided or overridden by other value.

4.3.1 State-element Level Primitives

We introduce the basic primitive of the model: **Create-se**. Prior to its introduction, we introduce three system functions that are used by it.

legal-temporal(t_v, t_d) is a boolean function that returns "true" if the predetermined temporal constraints are satisfied. These temporal constraints are:

1. t_v is a legal temporal element (not empty, contains non intersecting interval);
2. t_d is a legal chronon (according to the application's granularity);
3. $t_d \leq now$ (*now* is the current chronon, read from the system's clock).

legal-type(val, p) is a boolean function that returns "true" only if *val* is in the domain of the property p.

associate$(se, \alpha.p.\beta)$ is a function that associates the state-element with a the bucket β in a variable of the property p of the object α. α denotes the object as identified by its identifier (primary key), this is translated to the *OID* using a translation function.

Create-se: creates a new state-element.

Syntax: *create-se* (*oid, p, β, val,* τ_d, τ_v).

Semantics: DB $\oplus$ *create-se* (*oid, p, β, val,* τ_d, τ_v) $\equiv$
($\neg$ legal-temporal(τ_v, τ_d) $\vee \neg$ legal-type(*val, p*))$\rightarrow$ abort $\otimes$
DB′ :=DB $\cup\{se\}$ | $se = (se\text{-}id, oid, val, \tau_x, \tau_d, \tau_v) \wedge se\text{-}id$=generate-se-id() $\otimes$
associate(se, $\alpha.p.\beta$) | α.Object-id = *oid.*

This primitive adds a single state-element *se* to the bucket β of a variable $\alpha.p$ (the instance of the property p in the object α), after checking if certain integrity constraints are satisfied. It consists of two phases: adding the state-element to the database (each state-element is a separate entity with a unique identity in the database), and associating it with a variable and a bucket. DB′ is the new database state. *se-id* and τ_x are generated by the system; *se-id* is generated according to the object-identifiers' generation conventions [CK86]; the τ_x (transaction time) is determined at commit time.

For example: The operation *create-se(oid=864545, p=Patient-Name, β=data, val=Dan Cohen,* τ_d*=Dec 12 1993; 10:00pm,* τ_v*=[Dec 12 1993; 10:00pm,* ∞*))* applied in a transaction that committed on Dec 12 1993; 10:02pm, resulted in the state-element (s4) in Figure 5.

4.3.2 Variable Level Update Primitives

This section presents the semantics of the variable level primitives. To provide upward compatibility for non-temporal databases and to provide a shortcut for the standard cases and ease the use, omission of the time values is allowed, and thus a default should be provided. We define τ'_d to be:

$$\tau'_d := \begin{cases} now & \text{if } \tau_d\text{=nil} \\ \tau_d & \text{otherwise} \end{cases}$$

That is, τ'_d is assigned a default value of *now* (the current chronon read from the system's clock of the transaction start time), only if no value has been provided for τ_d. This default can be adjusted by the DBA at the application initiation, to be either the start time of the transaction, or to be left as a null value, and be interpreted at retrieval time according to a user-requested interpretation (e.g., t_x whose value could not be used before commit time).

Set-var assigns a new value to a variable's data.

Syntax: *set-var* (*oid, p, val,* τ_d, τ_v)

Semantics: set-var (oid, p, val, τ_d, τ_v) $\equiv$
create-se(oid, p, data, val, τ'_d, τ'_v) | α.Object-id = oid $\wedge$

$$\tau'_v := \begin{cases} [now, \infty) \cap \mathrm{AR}(\alpha) - \mathrm{FR}(\alpha.p) & \text{if } \tau_v\text{=nil} \\ \tau_v \cap \mathrm{AR}(\alpha) - \mathrm{FR}(\alpha.p) & \text{otherwise} \end{cases}$$

The default value for τ_v, in this primitive, is [now, ∞). This default has been used by other researchers (e.g., [BZ82]) assuming that the value was not valid from -∞. This default is a natural extension of the update logic in conventional databases, where a new value replaces an older one as of the time it is inserted to the database.

The functions FR and AR have been defined in Section 4.1. AR returns the set of chronons in which a given object is active, and FR returns the chronons in which a given variable is frozen. The actual valid time (τ'_v) is derived by intersecting τ_v with the times in which the variable can be modified $AR(\alpha) - FR(\alpha.p)$ (the modifiable range). The modification of the valid time provided by the user, stems from considering a temporal database as a set of many conventional databases, each of which is valid in a single chronon. Consequently, an update that affects a valid time interval in a temporal database is, in fact, a set of several independent updates, where each update can either succeed or fail in a given valid time chronon. A similar approach, in different contexts, was taken in other works as well (e.g., [Sno87]).

For example, the operation *set-var* (*oid=864545, p= Social-Security-Number, val=02345678,* τ_d*=nil,* τ_v*=nil*), applied to the database on Dec 12, 1993; 11:30pm, results in the creation of state-element (s14) in Figure 5.

Freeze-var freezes a variable.

Syntax: *freeze-var (oid, p,* τ_d*,* τ_v*)*

Semantics: freeze-var (oid, p, τ_d, τ_v) $\equiv$
create-se(oid, p, modify-control,"frozen", τ'_d, τ'_v) | α.Object-id = oid $\wedge$

$$\tau'_v := \begin{cases} [now, \infty) \cap \mathrm{AR}(\alpha) & \text{if } \tau_v\text{=nil} \\ \tau_v \cap \mathrm{AR}(\alpha) & \text{otherwise} \end{cases}$$

The default value for τ_v in this primitive is [now, ∞).

The actual valid time (τ'_v) is derived by intersecting τ_v with the activity range of the object.

For example, the operation *freeze-var (oid=864545, p= Social-Security-Number,* τ_d*=nil,* τ_v*=nil)*, applied to the database on Dec 12, 1993; 10:00pm, results in the creation of state-element (s6) in Figure 5.

Unfreeze-var unfreezes a given variable.

Syntax: *unfreeze-var* (*oid, p,* τ_d*,* τ_v)

Semantics:
unfreeze-var (oid, p, τ_d, τ_v) $\equiv$
create-se(oid, p, modify-control, "changeable", τ'_d, τ'_v) | α.Object-id = oid $\wedge$

$$\tau'_v := \begin{cases} [now, \infty) \cap \mathrm{AR}(\alpha) & \text{if } \tau_v\text{=nil} \\ \tau_v \cap \mathrm{AR}(\alpha) & \text{otherwise} \end{cases}$$

τ'_v is not calculated with respect to the frozen range of the variable. Thus, an *unfreeze-var* operation can override an earlier *freeze* decision.

For example, the operation *unfreeze-var* (*oid=864545, p= Record-Number,* τ_d*=nil,* τ_v*=nil*), that was applied to the database on Dec 13 1993; 10:02pm, resulted in the generation of state-element (s19) in Figure 5.

4.3.3 Object Level Update Primitives

Create-obj: creates a new object that is an instance of a given class.

Syntax: *oid := create-obj (class)*

Semantics:
oid := create-obj (c) ≡
oid := generate-obj-id() ⊗ create-se-oid (oid) ⊗ create-se-class-ref (c, oid).

create-se-oid(oid) ≡ create-se (oid, p="object-id", data, oid, τ_d, τ_v)| τ_d=now ∧ τ_v=[now, ∞)

create-se-class-ref (c, oid) ≡ create-se (oid, p="class-ref",data, c, τ_d, τ_v) | τ_d=now ∧ τ_v=[now, ∞).

The *create-obj* primitive creates two new state-elements. The first state-element designates an object identity; the object identity is generated by the database and is returned as a result of applying the *generate-obj-id* built-in function. The object identity is a frozen state-element, the frozen status is protected by a meta-data integrity constraint that prevents the change of its status. The second state-element is a reference to the class c that is given as an argument using the object-id that was created earlier. The values of the time types of both state-elements are generated by the system and represent the systems defaults. They **do not** represent the object's valid-time activespan, i.e., the time during which the object exists in the modeled reality. The activespan of an object is explicitly controlled by the user, and is associated with the *Object-Status* variable.

For example, the operation *create-obj* (*Patient*), that was applied to the database on Dec 12 1993; 10:00pm resulted in the generation of state-elements (s1) and (s2) as presented in Figure 5 and returns the value 864545.

Set-obj-status changes the object status in a given valid-time temporal element. Possible values of the object status are *Active, Suspended* and *Disabled*. *Object-Status* is a special variable that cannot be handled by regular variable operations, thus it has its own set of operations that includes **Set-obj-status** to set the value, and **freeze-obj-status** and **unfreeze-obj-status** to freeze and unfreeze this status, respectively.

Syntax: *set-obj-status* (*oid, sval,* τ_d, τ_v)

Semantics:
set-obj-status (oid, sval, τ_d, τ_v) ≡
create-se (oid, "object-status", data, sval, τ'_d, τ'_v) | α.Object-id = oid ∧
sval ∈ {"active", "suspended", "disabled"} ∧

$$\tau'_v = \begin{cases} [now, \infty) - \mathrm{FR}(\alpha.Object\text{-}Status) & \text{if } \tau_v\text{=nil} \\ \tau_v - \mathrm{FR}(\alpha.Object\text{-}Status) & \text{otherwise} \end{cases}$$

τ'_v has a default value of the temporal element [*now*, ∞). τ'_v and τ'_d determine the object's valid-time activespan.

For example, the operation *set-obj-status (oid=864545, sval="Active", τ_d=nil, τ_v=nil)*, applied to the database on Dec 12 1993; 10:00pm, results in the generation of state-element (s3) in Figure 5.

Freeze-obj-status freezes the object status in a given interval.

Syntax: *freeze-obj-status (oid, t_d, t_v)*

Semantics:
freeze-obj-status (oid, τ_d, τ_v) $\equiv$
freeze-var (oid, "Object-Status", τ_d', τ_v').

τ_v' has a default value of the temporal element $[now, \infty)$. τ_v' and τ_d' are used to determine the object's valid-time activespan.

For example, the operation *freeze-obj-status (oid=864545, t_d=nil, t_v= [Dec 12 1993; 10:00pm, ∞)*, applied to the database on Aug 25 1994; 8:15am, resulted in the generation of state-element (s23) in Figure 5. This operation freezes the object status retroactively during its entire activespan.

Unfreeze-obj-status Unfreezes the variable *Object-Status*.

Syntax: *unfreeze-obj-status* (*oid, τ_d, τ_v*)

Semantics:
unfreeze-obj-status (oid, τ_d, τ_v) $\equiv$
unfreeze-var (oid, "Object-Status", τ_d, τ_v).

Disable-Obj changes the object status to *Disabled* in the interval $[t_s, \infty)$, where t_s is the start time associated with the valid time, given as a parameter by the user. Only the start time of the interval is used since this status is final in the sense that the object can never be revived again. Consequently, the end chronon is set to ∞.

Syntax: *disable-obj* (*oid, τ_d, τ_v*)

Semantics:
disable-obj (oid, τ_d, τ_v) $\equiv \alpha$.Object-id = oid $\wedge$
$\tau_v' := \begin{cases} [now, \infty) - FR(\alpha.Object\text{-}Status) & \text{if } \tau_v\text{=nil} \\ [t_s, \infty) - FR(\alpha.Object\text{-}Status) & \text{otherwise} \end{cases} \otimes$
$\tau_v' \neq [t_s', \infty) \rightarrow$ abort $\otimes$
Set-obj-status (oid, "disabled", τ_d', τ_v')

τ_v' receives a default value of $[now, \infty)$. The *disable-obj* operation assumes that the object is disabled as of a certain chronon to infinity. If the object status is frozen at some chronon during the interval of the *disable-obj* operation, then the object-status cannot be changed in this chronon. Thus, the *disable-obj* operation cannot be completed and the transaction should be either aborted or treated as an exception.

For example, the result of the operation *disable-obj* (*oid=864545, τ_d=nil, τ_v=nil*), applied to the database on Aug 25 1994; 8:05am, is the same as the *freeze-obj* operation, as given above.

In the general case, the *disable-obj* operation is not reversible. However, in exceptional cases, an authorized DBA can use the *unfreeze-obj-status* to reverse the *disable-obj* operation and "rescue" the object.

4.4 Update Operation Types

The update operation types that have been discussed in Section 2 are defined using the primitives of Section 4.3. These update operation types are the only ones that are accessible to users.

Insert :

Syntax: *insert* $(c, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\}) \mid \nu_i$=$(p_i,\ val_i,\ \tau_{di},\ \tau_{vi})$.

Semantics:
insert (c, τ_{dl}, τ_{vl}, $\{\nu_1, \ldots, \nu_n\} \mid \nu_i$=$(p_i, val_i, \tau_{di}, \tau_{vi}) \equiv$
(exists-identifier (c, $\{\nu_1, \ldots, \nu_n\}$) $\rightarrow$ abort $\otimes$
oid := create-obj (c) $\otimes$
set-obj-status (oid, "active", τ_{dl}, τ_{vl}) $\otimes$
set-var (oid, p_1, val_1, τ_{d1}, τ_{v1}) $\otimes \ldots \otimes$
set-var (oid, p_n, val_n τ_{dn}, τ_{vn})

exists-identifier is a function, it takes as an argument a class id and the set of input variables, according to the class definition determines the object identifier (primary key) and checks if there exists an instance of the class c with the given identifier. If this function returns *true* then the transaction should abort.

oid is set to be the new object's id, using the *create-obj* operation. The *insert* operation creates the object, using *create-obj*, sets its status to be active, using *set-obj-status* and then updates its variables, using *set-var*. τ_{dl} and τ_{vl} are the decision and valid times of the object's valid-time activespan. i.e., the temporal element in which the object is active. The generated *oid* is returned to the user.

Example: A new patient is inserted to the database. The following operation provides the patient's name.
insert (c=*Patient*, τ_{dl}=*Dec 12 1993; 10:00pm*, τ_{vl}=[*Dec 12 1993; 10:00pm*, ∞)
$\{\nu_1$=(p_1=*Patient-Name*, val_1=*Dan Cohen*, τ_{d1}=*nil*,
τ_{v1}=[*Dec 12 1993; 10:00pm*, ∞))})

(s1)-(s4) of Figure 5 are the state-elements added to the database as a result of this operation.

Modify :

Syntax: *modify* $(c, obj, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\}) \mid \nu_i = (p_i, val_i, \tau_{di}, \tau_{vi})$.

Semantics:
modify (c, obj, τ_{dl}, τ_{vl}, $\{\nu_1, \ldots, \nu_n\} \mid \nu_i$=$(p_i, val_i, \tau_{di}, \tau_{vi})) \equiv$

oid := identify-obj (c, obj) $\otimes$
(oid = nil) $\rightarrow$ abort $\otimes$
($\tau_{vl} \neq$ nil) $\rightarrow$ set-obj-status (oid, "active", τ_{dl}, τ_{vl}) $\otimes$
set-var (oid, p_1, val_1, τ_{d1}, τ_{v1}) $\otimes \ldots \otimes$
set-var (oid, p_n, val_n, τ_{dn}, τ_{vn})

The *modify* operation retrieves the object identity, based on an identifier given by the user, using *identify-obj*. If the user assigns a value to the t_{vl}, then it resets the object's valid-time activespan. Finally, it updates its variables, using *set-var*. c denotes the class-id of the object.

identify-obj is a function that converts object-identifiers (primary keys) to object-identities (surrogates). If the sought object does not exist in the database, then the modify operation cannot be completed and the transaction should be either aborted or treated as an exception. If there is more than one qualifying object with the same object-identifier, then the user is prompted to decide which object is the required one.

Example: The operation
modify (*c=Medical-Record, obj=12345678-1,* τ_{vl}*=nil,* τ_{dl}*=nil,*
{ν_1=(p_1=*Disorder,* val_1=*partial treatment*[7] *,* τ_{d1} *=Dec 12 1993; 11:15pm,*
τ_{v1}=[*Dec 12 1993; 11:15pm,* ∞))})
changes one of the disorder's alternatives in the Diagnosis. It generates the state-element (s10) in Figure 5.

Suspend :

Syntax: *suspend* (*c, obj,* τ_{dl}*,* τ_{vl}).

Semantics:
suspend (c, obj, τ_{dl}, τ_{vl}) $\equiv$
oid := identify-obj (c, obj) $\otimes$
(oid = nil) $\rightarrow$ abort $\otimes$
set-obj-status (oid, "suspended", τ_{dl}, τ_{vl}).

The suspend operation generates a new state-element of the variable *Object-Status* with the value "suspended," using *set-obj-status*. The operation uses the object identity that is given by the *identify-obj* function.

For example, the following operation suspends the patient Dan Cohen as an active patient in the emergency room. As a result, state-element (s20) of Figure 5 is added to the database.
suspend (*c=Patient, obj=Dan Cohen,* τ_{dl}*=Dec 19 1993; 8:00am,*
τ_{vl}=[*Dec 19 1993; 8:00am,* ∞)).

Resume :

Syntax: *resume* $(c, obj, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\}) \mid \nu_i = (p_i, val_i, \tau_{di}, \tau_{vi})$.

[7]The medical term *partial treatment* refers to cases in which a treatment has not been completed, for example: a patient has failed to take the entire quantity of antibiotics assigned to him.

Semantics:
resume $(c, obj, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\} \mid \nu_i = (p_i, val_i, \tau_{di}, \tau_{vi}) \equiv$
oid := identify-obj (c, obj) $\otimes$
(oid = nil) $\rightarrow$ abort $\otimes$
set-obj-status (oid, "active", τ_{dl}, τ_{vl}) $\otimes$
set-var (oid, p_1, val_1, τ_{d1}, τ_{v1}) $\otimes \ldots \otimes$
set-var (oid, p_n, val_n, τ_{dn}, τ_{vn})

For example, the following operation resumes the patient Dan Cohen as an active patient when he is admitted again to the emergency room.
resume (*c=Patient, obj=Dan Cohen,* τ_{dl}*=Aug 24 1994; 12:00am,* τ_{vl}*=[Aug 24 1994; 12:00am,* ∞*))*
As a result, state-element (s21) of Figure 5 is added to the database.

Disable :

Syntax: *disable* (c, *obj*, τ_d, τ_v)

Semantics:
disable (c, obj, τ_d, τ_v) $\equiv$
oid := identify-obj (c, obj) $\otimes$
(oid = nil) $\rightarrow$ abort $\otimes$
disable-obj (oid, τ_d, τ_v)

In non-temporal databases, when an object is deleted, its information is removed from the database. In temporal databases, historical information is kept and the user can retrieve the contents of each object that was disabled, during its activity range. Moreover, modifications to the state of the object at times before it was disabled are allowed. For example, it is possible to retroactively update a medical record in the period it was open, during the time in which the record is already closed. The semantics of the *disable* operation is compatible with the "real world semantics," since it is possible that new information is discovered after an object is no longer in the active domain.

Freeze :

Syntax: *freeze* $(c, obj, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\}) \mid \nu_i = (p_i, \tau_{di}, \tau_{vi})$.

Semantics:
freeze (c, obj, τ_{dl}, τ_{vl}, $\{\nu_1, \ldots, \nu_n\} \mid \nu_i$=$(p_i, \tau_{di}, \tau_{vi}) \equiv$
oid := identify-obj (c, obj) $\otimes$
(oid = nil) $\rightarrow$ abort $\otimes$
($\tau_{vl} \neq$ nil) $\rightarrow$ freeze-obj-status (oid, τ_{dl}, τ_{vl}) $\otimes$
freeze-var (oid, p_1, τ_{d1}, τ_{v1}) $\otimes \ldots \otimes$ freeze-var (oid, p_n, τ_{dn}, τ_{vn})

A freeze operation can be applied to a single chronon, to an interval or to the entire variable history. This operation can be applied to non-temporal databases as well, such that a freeze operation always refers to the current state.

For example, the following operation freezes the *Social-Security-Number* of *Dan Cohen*
freeze (*c=Patient, obj=Dan Cohen,* t_{dl}*=nil,* t_{vl}*=nil,* $\{v_1$ *=(*p_1*=Social-Security-Number,*

t_{d1}=*Dec 12 1993; 10:00pm,* t_{v1}=*[Dec 12 1993; 10:00pm,* ∞*))*})
As a result, state-element (s6) of Figure 5 is added to the database.

Unfreeze :

Syntax: *unfreeze* $(c, obj, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\}) \mid \nu_i = (p_i, \tau_{di}, \tau_{vi})$.

Semantics:
unfreeze (c, obj, τ_{dl}, τ_{vl}, {ν_1, ..., ν_n}) | ν_i=(p_i, τ_{di}, τ_{vi}) ≡
oid := identify-obj (c, obj) ⊗
(oid = nil) → abort ⊗
($\tau_{vl} \neq$ nil) → unfreeze-obj-status (oid, τ_{dl}, τ_{vl}) ⊗
unfreeze-var (oid, p_1, τ_{d1}, τ_{v1}) ⊗ ... ⊗
unfreeze-var (oid, p_n, τ_{dn}, τ_{vn})

An unfreeze operation eliminates the "freeze" constraint (if it exists) for the specified valid time. For example, the following operation unfreezes the *Record-Number* variable.
unfreeze (*c=Medical-Record, obj=12345678-1,* τ_{dl}*=nil,* τ_{vl}*=nil,*
{ν_1 *=(*p_1*=Record-Number,* τ_{d1}*=Dec 13 1993; 10:00pm,* τ_{v1}*=[Dec 13 1993; 10:00pm,* ∞*))*})
As a result, state-element (s19) of Figure 5 is added to the database.

Revise :

Syntax: *revise* $(c, obj, \tau_{dl}, \tau_{vl}, \{\gamma_1, \ldots, \gamma_n\}) \mid \gamma_i = (\nu_i, sq_i), \nu_i = (p_i, val_i, \tau_{di}, \tau_{vi})$

Semantics:
revise $(c, obj, \tau_{dl}, \tau_{vl}, \{\gamma_1, \ldots, \gamma_n\}) \mid \gamma_i = (\nu_i, sq_i), \nu_i = (p_i, val_i, \tau_{di}, \tau_{vi}) \equiv$
oid := identify-obj (c, obj) ⊗
(oid = nil) → *abort* ⊗
($\tau_{vl} \neq$ *nil)* → *set-obj-status (oid, "active",* τ_{dl}*,* τ_{vl}*)* ⊗
$val_1 \neq$ *nil* → *modify* $(c, obj, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\})$⊗
$\forall se_i \in sq_1 \cup \ldots \cup sq_n$*: create-se(oid,* p_i*, void-SE,* $se_i, \tau_{di}, \tau_{vi}$).

The **revise** operation replaces existing values with new ones, voiding the old values. Each revised value may cause the revision of multiple state-elements, selected by a selection query sq_i. A revise operation can affect more than one state-elements in the following cases:

1. The valid time of the correction covers the valid time of several existing state-elements.
2. A change from a multi-valued semantics to a unique value semantics requires to void several state-elements.

The **revise** operation has two parts. The first part adds state-elements with new values if there is at least one value that is not nil. If this part is not activated, then the state-elements are voided without replacing them with new values; the second part uses a selection query sq_i for each revised value, to locate the state-elements that should be voided, and voids these state-elements, or any part of their validity time that is specified by the τ_v variable.

For example, the operation *revise* (*c=Patient, obj=Dan Cohen,* $\{\nu_1$ *=*(p_1*= Social-Security-Number,* val_1*=02345678,* τ_{d1}*=Dec 12 1993; 11:30pm,* τ_{v1}*=*[*Dec 12 1993; 11:30pm,* ∞*),* sq_1*=* **select the state-element with value=12345678**) applied in a transaction that committed at Dec 12 1993; 11:33pm, resulted in the creation state-elements $(s13), (s14)$ in Figure 5.

The revise operation allows the replacement of a frozen value, marking it as an erroneous one. The *revise* operation is necessary, along with the *modify* operation, in order to make a semantic distinction between a change in the real world and between a correction of a wrong value that was reported to the database. The default retrieve operations exclude revised values in retrieval operations (this default can be overridden). Additional use of the *revise* operation is to void state-elements without replacing them. In this case, $\nu_i = nil$ and only the second part of the revise operation is applied.

Set-SVS : Syntax: *set-SVS* $(c, obj, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\}) \mid \nu_i = (p_i, svs_i, qid_i, \tau_{di}, \tau_{vi})$

semantics: *set-SVS* $(c, obj, \tau_{dl}, \tau_{vl}, \{\nu_1, \ldots, \nu_n\}) \mid \nu_i = (p_i, svs_i, qid_i, \tau_{di}, \tau_{vi}) \equiv$
oid := identify-obj (c, obj) $\otimes$
(oid = nil) $\rightarrow$ *abort* $\otimes$
$(\tau_{vl} \neq nil) \rightarrow$ *set-obj-status(oid, "active",*τ_{dl}, τ_{vl}*)* $\otimes$
create-se (oid, p_1*, variable-SVS,* (svs_1, qid_1)*,* τ_{d1}, τ_{v1}*)* $\otimes, \ldots, \otimes$
create-se (oid, p_1*, variable-SVS,* (svs_1, qid_n)*,* τ_{dn}, τ_{vn}*)*

The set-SVS command sets the SVS interpretation of one or more variables that belong to the same object. The interpretation consists of two parts: the SVS keyword (**first, last, all, single, multi**) and a query id. A query id is meaningful only when the **single** or **multi** keywords are used, otherwise it is ignored.

4.5 Discussion

The update operation types are used as a uniform linguistic abstraction that supports any type of database update, for the data and control parts. The **Insert** operation type creates a new object, it can also update the data bucket of the variables in the created object. The **Modify** operation updates the data bucket of the variables in an existing object. Their semantics are an extended version of the semantics of these operation types in regular databases. The extended semantics follow the temporal database's structure. These operations are implemented using the **set-var** operation. The **Suspend, Resume**, and **Disable** operation are operations that affect the object-status. The **Resume** operation can also be used to update the data bucket of the variables that belong to the object it resumes. The **Freeze** and **Unfreeze** update the *modify-control* buckets of variables that belong to a given object, they use the **freeze-var** and **unfreeze-var** operations. The **Revise** operation updates the data bucket of the revised variable, and marks the revised state-elements in the Void-SE bucket. Note that the **Revise** semantics does not use **set-var**, but instead it uses state-elements operations directly. This is done to bypass the **frozen** constraint, if exists, because it is possible to revise any state-element, even if it's variable is frozen. The **Set-SVS** sets SVS interpretation that overrides the static interpretation in the variable's level.

This set of update operation types is a minimal set, but it is not necessarily the set that is appropriate for each application. It is possible to eliminate certain operations (i.e., not allow the **Revise** operation, in applications that do not support revisions) or to construct new operations using the low level primitives. For example, the combination of **Modify** and **Freeze** in a single operation would enable to update values and freeze them using a single linguistic primitive. A formal definition of a new update operation type can be based on the predefined low-level primitives and should consider the following issues:

1. Whether the update operation type is applied with respect to the frozen range of the variable, the changeable range of the variable, or both?
2. What are the appropriate defaults for t_v and t_d?
3. What are the constraints whose violation lead to a transaction failure?

5 Implementation Issues

Several implementation issues are discussed in this section. Section 5.1 discusses alternatives for implementing the additional functionalities. Section 5.2 discusses the implementation of **decision time** as a primitive, Section 5.3 discusses the implementation in a temporal relational model, Section 5.4 discusses the mapping of the proposed model to TSQL2, and proposes some changes to TSQL2 in order to facilitate the support for the extended functionality.

5.1 The Implementation Alternatives

The functionality discussed in this paper does not exist in TSQL2 or in any other temporal language, at the primitive level. The implementation alternatives are:

- using the proposed primitives as system design tools, using the existing database primitives at the database implementation level;
- developing a *wrapper* based on the temporal infrastructure, whose primitives are compatible with the primitives presented in this paper;
- devising a separate implementation model bypassing the temporal database infrastructure, for the use of applications that require the extended functionalities.

The first alternative cannot satisfy this study's objectives; the use of the existing primitives would make writing programs that satisfy these extended functionalities tedious, hard to

verify, and ad-hoc. The third alternative of devising a new implementation model is consistent with our objectives and can result in optimized performance. The construction of a standard model that combines the desired object oriented and temporal features is a major task for future research and development in the temporal database community in general, and we intend to base our further implementation on such a model. Our current prototype implementation is based on a relational database, using a subset of TSQL2. In general, we propose to implement our model as a wrapper on top of an TSQL2 implementation.

5.2 The Implementation of the Decision Time Primitive

The following discussion is relevant for applications that require the decision time functionality. We argued that the decision time in some applications is indispensable in determining the correct order of real-world events, and in making decision analysis inferences. The implementation choices are whether to implement *decision time* as an additional time dimension, or try to achieve this functionality in another way. The decision time has two major impacts on the model's representation and semantics:

- It adds an additional chronon to each state-element;
- The function ASE that selects the valid value according to the **last value** SVS may employ the decision time and not the transaction time to determine the last value (the same may apply for **first value** SVS).

It is possible to emulate the decision time functionality, without using an explicit time type, by adding objects that designate decisions,[8] and using the beginning of the t_v interval of their variables to denote decision time. Such a solution is proposed in [OS95], and it complies with the desire not to add more primitives. However, we argue that this requirement is general, and important enough to have a direct representation. Using decision-time objects is too cumbersome, even at the logical level.

- From space complexity point of view, adding the decision time to the state-element level requires substantially less space than the creation of a redundant object;
- From time complexity point of view, having the decision time available at the state-element level is less expensive than joining the state-element and the decision-related object;
- From the development and maintenance point of view, it is clearer to the user since the decision-related object is not a concrete object in the application's domain.

[8] Recall that 'decision' is a generic reference to the real-world event that led to the database transaction. In many applications it represents an actual decision.

This analysis leads to the conclusion that the support of decision time as a model primitive is cost effective in cases that this functionality is required.

If the decision time functionality is not required, we may eliminate the space overhead, by supporting an initialization parameter that eliminates the decision time. In this case, the decision time support is an optional feature selected at the initialization time of the application's schema. If decision time is not selected, then no space is saved for t_d at the state-element's level and the transaction time (t_x) replaces the decision time (t_d) in the interpretation of the ASE retrieval function. An existing application can be converted to include decision time, such that the value of t_x will be used for any missing value of t_d.

5.3 Implementation of the Model

The update functionality presented in this paper is "data model independent" in the sense that it can be implemented on various data models. Although a natural implementation is in an object-oriented model, a standard object oriented temporal data model does not exist. We therefore restrict this discussion to the relational model and TSQL2. The structure defined in this paper can be trivially mapped into the nested relational model [RKS88] that has been suggested in [Tan86] to be a basis for temporal database implementation. Mapping the data structure into a "flat" relational model requires the use of normalization rules. The implementation in the temporal model is not unique. A possible implementation can use universal relations as discussed in [N$^+$87]. Another possible implementation uses the ENF (Extension Normal Form), which is an extension of the TNF (Time Normal Form) [NA89], as follows.

Each relation designates a set of *synchronous attributes*, which are attributes that have common state-element's additional information (t_d, t_v, etc.) at any chronon. We extend the definition of TNF to include all the additional information in a state-element, rather then just the $t_v$that represents an atomic combination of property and bucket consists of values and the state-element's additional information (without the *revised-se* component). A state-element of a set property is represented by several tuples with the same *se-id*. Each tuple is identified by both the *se-id* and its value.

The implementation using the ENF blurs the original schema structure. Thus, the relationship among a class and its properties is represented using an additional relation for each bucket. Another relation stores the classification of objects into classes. Each relation represents a single combination of property and bucket, with the state-element's additional information. Note that in this particular example, all attributes are asynchronous. *Object-id-data*, *Class-ref-data*, and *Object-status-data* are system variables. *Treatment-Data* is a user defined Property.

The creation of a state-element involves the addition of new tuple(s) to the appropriate property-bucket relation.

This representation is restricted since the t_v can be an interval but not a temporal element. To eliminate this restriction, a separate relation for the t_v element should be created, identified by the state-element-id and the interval values.

Redundant timestamping exist in t_x when multiple state-elements are updated in the same transaction, in t_v, when multiple state-elements have the same valid time, and in t_d, when multiple state-elements have the same decision times. Furthermore, there can be an overlap among all of them, e.g., $t_x = t_d = t_s(t_v)$. A possible space optimization feature is the enumeration of chronons and its use instead of the full representation; this, however, requires a conversion table from the enumeration to time-stamps, increasing the retrieval time.

5.4 Supporting The Extended Update Functionality With TSQL2

In this section we present a mapping of the update functionality to TSQL2, and the additional clauses required to augment TSQL2

5.4.1 Mapping to TSQL2

TSQL2 supports a bitemporal database environment, and uses temporal clauses as an extension of SQL-92.

It is sufficient to map the *create-se* operation. The rest of the operations are translated to *create-se* as shown in Section 3. For the translation we assume that we have an underlying ENF bitemporal relational database with TSQL2 embedded within a host language that controls integrity constraints and aborts the transaction when necessary.

create-se(oid, p, β, val, τ_d, τ_v, s)≡INSERT INTO p-β
VALUES (NEW, val, oid, τ_d, s)
VALID TIMESTAMP τ_v

oid is the object id. *val* is the set of all values that have a common state-element's additional information (e.g., t_v, t_d, etc.) *s* can be either "changeable" or "frozen." Since t_v is part of the temporal infrastructure schema, it is updated using the TSQL2 feature *VALID TIMESTAMP* and not as part of the *VALUES* clause.

The retrieval selections CSE and ASE can be easily expressed by TSQL2 queries as well.

5.4.2 A Proposal to Augment TSQL2

In order to support that functionality described in this paper in a convenient way, the following features should be supported as a primitive level, this can be done as a shell on

top of TSQL2[S+94], or as direct extension to the TSQL2 language.

1. A mechanism for handling simultaneous values is required. This mechanism should include new functions and defaults to support the retrieval of simultaneous values. These functions consist of the CSE and ASE functions.

2. A third time type (decision time) that reflects the correct order of occurrences in the modeled reality is needed.

3. A mechanism for freezing object's values and enforcing freezing constraints should be added.

4. A correction operation, that is semantically distinct from modification, should be introduced.

5. Clauses that represent the functionality of the update operation types would make the update language more powerful. We suggest to include the following new clauses in the extension of TSQL2.

   SUSPEND p: This clause would have a similar effect as the *suspend* primitive presented in Section 4. The *disable* primitive can use the semantics of the DELETE clause of TSQL2. The use of *delete* as an alias to *disable* is necessary to guarantee the compatibility of TSQL2 with SQL-92.

   It should be noted that TSQL2 permits changes to existing tuples even after the transaction commits. This can prevent the ability to restore all past states of the database. For example, a DELETE operation in a bitemporal database changes the t_x according to the parameter given in this clause. Since deletions can be proactive and retroactive as well as current, the time of issuing the DELETE operation is not known after the modification. Consequently, queries with a viewpoint earlier than the time of change cannot be answered.

   RESUME p: This clause would have a similar effect as the *resume* operation type that was presented in Section 4. The clause:

   RESUME p
   VALID TIMESTAMP τ_v

   would effect an existing tuple, and change its validity interval.

   FREEZE: This clause would have a similar effect as the *freezing* operation type, as presented in Section 4. For example,

   FREEZE VARIABLES (var_1, ..., var_n) OF p
   VALID TIMESTAMP τ_v

   The FREEZE clause would *freeze* a set of a variables in a given valid time interval, but it can be effective only when it is combined with a mechanism that enforces the frozen range of a variable.

UNFREEZE: This clause would have a similar effect as the *unfreeze* operation type that was presented in Section 4. For example, the clause:

UNFREEZE VARIABLES (var_1, ..., var_n) OF p
VALID TIMESTAMP τ_v

would *unfreeze* a set of variables, in a given valid time interval.

REVISE...WITH: This clause would have a similar effect as the *revise* operation type that was presented in Section 4. For example, the clause:

REVISE *se*
WITH VALUE (val)

would *revise* the state-element *se* with the value with the value val.

SET-SVS This clause would have a similar effect as the *set-svs* operation type that was presented in Section 4. For example, the clause:

SET-SVS *var*
WITH VALUE (val)
USING QUERY (qid)

would set the SVS value and associated query of var.

6 Conclusion

This work extends the temporal database functionality to accommodate complex applications. These extensions are a step in the direction of bridging the gap between temporal database capability and the needs of real-life applications. The results presented in this paper support a model that support flexible interpretation of simultaneous values semantics as an integral part of a temporal database. This functionality facilitates the database modeling and manipulation of real-world concepts.

The main contribution of this paper is in the construction of a model that supports extended update features in both the schema level and update operation levels. The features include: simultaneous values semantics, modify control and revision control, all of them are required due to the simultaneous values capability of temporal databases. The case study has exemplified the need for such a model in a decision analysis system, however these functionalities can be used for other types of system. For example, it can be used to tailor a data model's capabilities according to application's needs, by adjusting the meta-data property *class-ref* property single or multiple classification of an object and fixed or variable classification of objects.

The model presented in this paper includes a third time type called *decision time* that maintains the correct order of events in the modeled reality. This time type is essential for

many types of applications, and is optionally supported as a model primitive. The system designer can choose if this feature is included, during the application's initialization time.

The proposed update functionality is data model independent, and thus it can be designed as a shell on top of existing data models. A mapping of the update primitives to TSQL2 was described, as well as a list of extensions to TSQL2 required for a more complete temporal database functionality.

A prototype of this system is currently being developed. This prototype is to be used in a simulation project in a hospital's training center. Further research will deal with data modeling implementation on top of an object oriented model, the impact of simultaneous values on schema versioning, and investigation of applying research that has been done in the artificial intelligence area about possible world semantics and belief revision to extend this model.

Acknowledgments

The case study was established with the help of Gilad Rosenberg M.D. We thank the reviewers for many helpful comments.

References

[ABN87] T. Abbod, K. Brown, and H. Noble. Providing time-related constraints for conventional database systems. In *Proceedings of the International conference on very Large Data Bases (VLDB)*, pages 167–175, Brighton, 1987.

[AKG91] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78, 1991.

[Ari86] G. Ariav. A temporally oriented data model. *ACM Transactions on Database Systems (TODS)*, 11(4):499–527, December 1986.

[Bra78] J. Bradely. Operations in databases. In *Proceedings of the International conference on very Large Data Bases (VLDB)*, W. Berlin, 1978.

[BZ82] J. Ben-Zvi. *The Time Relational Model*. PhD thesis, Computer Science Department, UCLA, 1982.

[CC87] J. Clifford and A. Crocker. The historical relational data model (hrdm) and algebra based on lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537, Feb 1987.

[CK86] G.P. Copeland and S. Khoshafian. Object identity. In *Proceedings of Object Oriented Programming Systems, Languages and Applications*. ACM, 1986.

[CK94] S. Chakravarthy and S.-K. Kim. Resolution of time concepts in temporal databases. *Information Sciences*, 80(1-2):43–89, Sept. 1994.

[CT85] J. Clifford and A. U. Tansel. On an algebra for historical relational databases: two views. In *Proceedings of the ACM SIGMOD*, pages 247–265, May 1985.

[D+79] V. Deantonellis et al. Extending the entity-relationship approach to take into account historical aspects of systems. In *Proceedings of the International Conference on the E-R Approach to Systems Analysis and Design*. North Holland, 1979.

[EGS92] O. Etzion, A. Gal, and A. Segev. Temporal support in active databases. In *Proceedings of the Workshop on Information Technologies & Systems (WITS)*, pages 245–254, Dec 1992.

[EW90] R. Elmasri and G. Wuu. A temporal model and query language for ER database. In *Proceedings of the International Conference on Data Engineering*, pages 76–83, Feb 1990.

[F+94] R. Fagin et al. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, 1994.

[FD71] N. Findler and D.Chen. On the problems of time retrieval, temporal relations, causality and coexistence. In *Proceedings of the International Conference on Artificial Intelligence*. Imperial College, Sep 1971.

[Gad88] S.K. Gadia. The role of temporal elements in temporal databases. *Data Engineering Bulletin*, 7:197–203, 1988.

[GE98] A. Gal and O. Etzion. A multiagent update process in a database with temporal data dependencies and schema versioning. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(1):21–37, 1998.

[GES94] A. Gal, O. Etzion, and A. Segev. Representation of highly-complex knowledge in a database. *Journal of Intelligent Information Systems*, 3(2):185–203, Mar 1994.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, application and research issues. *ACM Computing Surveys*, 19(3):201–260, Sep 1987.

[J+94] C.S. Jensen et al. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–63, 1994.

[KL83] M.R. Klopprogge and P.C. Lockemann. Modeling information preserving databases; consequences of the concept of time. In *Proceedings of the International conference on very Large Data Bases (VLDB)*, Florance, Italy, 1983.

[Kli93] N. Kline. An update of the temporal database bibliography. *ACM SIGMOD Record*, 22(4):66–80, December 1993.

[McK88] E. McKenzie. *An Algebraic Language for Query and Update of Temporal Databases*. PhD thesis, Computer Science Department, University of North Carolina in Chapel Hill, Sep 1988.

[MS91] E. McKenzie and R. Snodgrass. An evaluation of relational algebras incorporating the time dimension in databases. *ACM Computer Surveys*, 23(4):501–543, Dec 1991.

[N+87] B.A. Nixon et al. Design of a compiler for a semantic data model. Technical Report CSRI-44, Computer Systems Research Institute, University of Toronto, May 1987.

[NA89] S.B. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49:147–175, 1989.

[OS95] G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4), August 1995.

[PSE+94] N. Pissinou, R.T. Snodgrass, R. Elmasri, I.S. Mumick, M.T. Ozsu, B. Pernici, A. Segev, and B. Theodoulidis. Towards an infrastructure for temporal databases—A workshop report. *ACM SIGMOD Record*, 23(1):35, 1994.

[RKS88] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems (TODS)*, 13(4):390–417, Dec 1988.

[RS91] E. Rose and A. Segev. Toodm-a temporal, object-oriented data model with temporal constraints. In *Proceedings of the International Conference on the Entity-Relationship Approach*, pages 205–229, San Mateo, California, 1991.

[S+94] R. Snodgrass et al. TSQL2 language specification. *ACM SIGMOD Record*, 23(1):65–86, Mar 1994.

[SA86] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19:35–42, Sep 1986.

[Sar93] 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.

[SJS95] A. Segev, C.J. Jensen, and R. Snodgrass. Report on the 1995 international workshop on temporal databses. *ACM SIGMOD Record*, 24(4):46–52, Dec 1995.

[SK86] A. Shoshani and K. Kawagoe. Temporal data management. In *Proceedings of the International conference on very Large Data Bases (VLDB)*, pages 79–88, Aug 1986.

[Sno87] R. Snodgrass. The temporal query language TQUEL. *ACM Transactions on Database Systems (TODS)*, 12(2):247–298, June 1987.

[Soo91] M.D. Soo. Bibliography on temporal databases. *ACM SIGMOD Record*, 20(1):14–24, 1991.

[SS88] A. Segev and A. Shoshani. The representation of a temporal data model in the relational environment. Technical Report LBL-25461, Lawrence Berkeley Laboratories, Aug 1988. Invited Paper to the 4th International Conference on Statistical and Scientific Database Management.

[Tan86] A.U. Tansel. Adding time dimension to relational model and extending relational algebra. *Information Systems*, 11(4):343–355, 1986.

[TCG+93] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases*. The Benjamin/Commings Publishing Company, Inc., Redwood City, CA., 1993.

[TK96] V.J. Tsotras and A. Kumar. Temporal database bibliography update. *ACM SIGMOD Record*, 25(1), March 1996.

[WJL91] G. Wiederhold, S. Jajodia, and W. Litwin. Dealing with granularity of time in temporal databases. In R. Anderson et al., editors, *Lecture Notes in Computer Science 498*, pages 124–140. Springer-Verlag, 1991.

[ZP93] E. Zimanyi and A. Pirotte. Imperfect knowledge in databases. In P. Smets and A. Motro, editors, *Proceedings of the Workshop on Uncertainty Management in Information Systems: From Needs to Solutions*, pages 136–186, Santa Catalins, CA., Apr 1993.