MODELING PERIODIC TIME, PERIODIC TEMPORAL DATABASES, AND CALENDARS

by

ATAKAN KURT

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Thesis Advisor: Professor Z. Meral Ozsoyoglu

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY

May 1996

MODELING PERIODIC TIME, PERIODIC TEMPORAL DATABASES, AND CALENDARS

Abstract

by

Atakan Kurt

Periodic temporal events occur in scheduling, planning, medical records, calendars, scientific, multimedia databases etc. A framework for describing and reasoning about periodic events by means of a new temporal type called *periodic element* in general is presented. Periodic elements (*i*) are capable of representing not only *aperiodic* and *strictly-periodic* temporal events but also *partially-periodic* events as a single temporal value. (*ii*) They can represent *absolute* and *relative* events uniformly (with conversions between the two), and (*iii*) are closed under the set theoretic operators. Our paradigm is user friendly in the sense that we don't introduce new operators or constructs but rather extend the existing operators of aperiodic types for periodic time.

Existing temporal models support only aperiodic events with a few exceptions. The ones supporting periodic time have a number of deficiencies. We introduce periodic time support into object oriented model by modeling temporal values as a function of time. Thus a temporal attribute is a method returning the snapshot value at a given time instant similar to time-slice operators. Various other temporal operators such as lifespan, transaction time of an object are implemented by methods. The periodic temporal data model supports both aperiodic and periodic time though temporal types of *instant, interval, temporal element*, and *periodic element*. The temporal types are implemented as abstract data types. The behavior of there types are defined by a set of functions of the set theoretic operators, temporal transformations, temporal comparisons and etc. These functions contribute towards a powerful temporal query language. Our paradigm is compatible with existing

aperiodic temporal data models, i.e., it can be used to extend them seamlessly to incorporate periodic time into the model, because it does neither modify the data model nor augment the query language with new temporal constructs.

Periodic elements can be used for modeling the time aspect of calendars, multimedia, and scientific applications too. To substantiate this we apply periodic elements to modeling calendars and calendric events, i.e., events involving calendars. We show how calendars can be represented by periodic elements and how complex calendars are derived from simpler ones by means of temporal transformations. Further we introduced the *periodic interval selection* operator and *periodic interval* construct both are natural extensions of their aperiodic counterparts for periodic events.

Table of Contents

1. Introduction **1**

1.1 Basic Concepts 4

1.2 Previous Work 5

1.3 Summary 7

2. Background **9**

2.1 Instant 11

2.2 Interval 12

2.3 Temporal Element 15

2.4 Summary 20

3. Modeling Periodic Time **21**

3.1 Periodic Element as a Temporal Type 22

3.1.1 Temporal transformations 23

3.1.2 Specialized Operators: 24

3.1.3 Formatting Functions 27

3.1.4 Set Theoretic Operators 35

3.1.5 Relational Operators 44

3.2 Relative Periodic Time 45

3.2.1 Relative Intervals 46

3.2.2 Relative Temporal Elements 47

3.2.3 Relative Periodic Elements 49

3.2.4 Conversion between Relative and Absolute time 49

3.3 Finite Periodic Time 53

3.4 Summary 56

4. Periodic Temporal Data Model **58**

4.1 Modeling Temporal Data Using Methods 60

4.2 Time as a Set of Abstract Data Types............62
4.3 Valid time............64
4.4 Temporal Constraints............67
4.5 Summary............68

5. Implementation............69

5.1 Temporal Types............70
5.1.1 General Purpose Temporal Functions............70
5.1.2 Set Theoretic Operators............71
5.1.3 Temporal Transformations............74
5.2 Temporal Objects............75
5.2.1 Data Structure for Valid Time............77
5.2.2 Attribute Level Temporal Methods............77
5.2.3 Object Level Valid Time Methods............78
5.2.4 Transaction Time Support............79
5.3 Temporal Queries............80

6. Modeling Calendars and Calendric Operations............87

6.1 Modeling Calendars............89
6.2 Derivation of Calendars............92
6.3 Calendric Events............94
6.4 Periodic Interval Selection............96
6.5 Periodic Intervals............99
6.6 Temporal Relationships between Calendric Events............101
6.7 Modeling Relative Events............102
6.8 Calendric Expressions............104
6.8.1 Calendar Conversion............104
6.9 Summary............105

7. Related Research and Comparison............106

7.1 TSQL2 106
7.2 Collection of Intervals 107
7.3 Generalized Relations 108
7.4 Calendars and Slices 110
7.5 Summary 112

8. Conclusion 114

9. Appendix 121

9.1 Types 121
9.2 Class Definitions 121
9.2.1 Methods 121
9.3 Instant 124
9.3.1 Specialized Functions 124
9.3.2 Temporal Transformations 125
9.3.3 Temporal Comparisons 125
9.4 Interval 126
9.4.1 Specialized Functions 126
9.4.2 Temporal Transformations 127
9.4.3 Temporal Comparisons 127
9.4.4 Set Theoretic Operators 128
9.5 Element 129
9.5.1 Specialized Functions 129
9.5.2 Temporal Transformations 131
9.5.3 Temporal Comparisons 132
9.5.4 Set theoretic Operators 134
9.6 Periodic Element 134
9.6.1 Specialized Functions 134
9.6.2 Temporal Transformations 142
9.6.3 Temporal Comparisons 143

9.6.4 Set Theoretic Operators 144

List of Figures

FIGURE 2.1: RELATIONAL OPERATORS. 14
FIGURE 2.2: APERIODIC TYPES AND PERIODIC ELEMENT. 15
FIGURE 2.3: RELATIONAL OPERATORS ON ELEMENTS. 17
FIGURE 2.4: THE SET THEORETIC OPERATORS ON TEMPORAL ELEMENTS. 19
FIGURE 3.1: PERIODIC ELEMENT. 23
FIGURE 3.2: TEMPORAL TRANSFORMATIONS OF PERIODIC ELEMENTS 24
FIGURE 3.3: FORMATTING FUNCTIONS ON PERIODIC ELEMENTS 28
FIGURE 3.4: THE SET THEORETIC OPERATORS ON PERIODIC ELEMENTS. 42
FIGURE 6.1: BASIC AND ARBITRARY CALENDARS. 91

Chapter 1

Introduction

Traditional database systems maintain the current state of data in the database. When data is modified, the old data is overridden with the new data. As opposed to traditional database systems, temporal databases store the past and the future data in addition to the current data. This allows users to query past (and possibly future) state of the data. Several temporal data models have been proposed extending relational, nested-relational, and object-oriented data models with a time dimension in the literature [BP 85, GV 85, T 86, CC 87, SS 87, G 88, EW 90, KSW 90,SC 91, S 91, T 91, RS 91, W 92, WD 92, A 93, LBG 93, T 93, GB 93, S 94, S 95]. There are also temporal extensions proposed for logic languages [CI 88, GB 89, AM 89, B 91, BCW 93]. One common point of these proposals with two exceptions [CI 88, KSW 90] is that they can maintain only aperiodic temporal events in the database. An aperiodic event could be an instantaneous event e.g. the departure time of a train, an event happening for an interval of time e.g. the travel time of a train between 2 stations, or the travel times of a train in a given week. These aperiodic events are modeled in the literature using the *instant*, *interval* and *temporal element* types respectively. However many applications such as scheduling, planning, forecasting, time-management, time-series, scientific databases, multimedia databases, active databases, banking, law, medical records, accounting, process control, inventory control deal with periodic events besides aperiodic ones. Periodic events can simply be defined as temporal events happening periodically. Since periodic events occur possibly infinite number of times, they cannot be represented by the aperiodic temporal types mentioned above which employs finite representations. Furthermore, periodic events have richer semantics and are inherently more complex than aperiodic events. Below are a few applications in which periodic events occur.

- Scheduling employees and tasks in an organization involves periodic time. Part-time employees may have daily, weekly or monthly work schedules. Even temporal templates (temporal events with no absolute time references) may be used as predefined timetables or schedules. For example, a temporal template may state that a part-time cashier is to attend 6-hour training sessions in the first 5 days of the employment and to work 4-hours for 3 successive-days every week.
- A multimedia database stores video and audio sequences which must be synchronously played out. The presentation of audio/video sequences may also be modeled as a periodic event.
- Active databases are characterized by the *on-event if-condition do-action* rules. These rules must be evaluated periodically to check if any of the conditions are satisfied. The *if-condition* clause may include calendric expressions such as *every 10 minutes*, *every Monday* or *the first day of every month*. Such calendric events also occur in business, medical, law applications. Natural language expressions involving calendars (called *calendric expressions*) like *the first Monday of every month*, *the fourth Thursday of Novembers*, *the work days in year 1995*, *every year between Thanksgiving and Christmas* are difficult to represent and manipulate, because of the richness of the semantics.

The temporal data models proposed so far deal exclusively with aperiodic temporal data to a great extent. The data models for handling periodic events are very rare and have a number of deficiencies. There is a need for defining a periodic data type for the application domains mentioned above [W 93, B 93]. Our objective is to model periodic events in temporal databases. Let us give an example for motivation.

Example 1.1

Imagine a company that employs full-time and part-time employees. Each part-time employee works according to a daily, or a weekly work schedule. The management wants to define a set of daily, or weekly work schedules to be assigned to the part-time employees based on the tasks to be performed by these employees. These work schedules, called temporal templates or templates for short, are to be defined

relatively, i.e., the definition of these templates should not contain any absolute dates or time instants, but they would rather be defined with respect to an unspecified beginning time. For example, employees in $task_1$ category go through a 5-day training session first, then start working 3 days later, and work every other day (Such tasks are often encountered in factories where tasks are pipelined in production lines). When a part-time employee is employed, an existing template is to be assigned to the worker. The management wants to store the schedule of the employee in terms of absolute time values, i.e., from the actual starting time of the employee, so that the number of actual hours he/she worked can be calculated.

Full-time employees work aperiodically. Every employee works in a department. Some employees are subject to periodical rotations among departments. For example, for training purposes some employees must switch to a different department in every 3 months or certain workers may be switched to the ice cream department in summer time to counter the increased demand. Among other conventional temporal queries (queries about the past), the management wants to use the periodic nature of events to plan the future activities, to estimate the sum of the salaries of workers in a specific time interval in the future, the number of part-time workers at a past or future date, the employees working in a department at a given future date, or on certain days like Mondays etc.

□

Periodic temporal applications are currently handled in an ad hoc manner which can lead to expensive application development and maintenance where periodic data is stored in non-standard data structures and queried through user programs only. Since each application uses its own data structure and operators, periodic data cannot be shared between different applications. The data models for handling periodic events [L 86, CI 88, KSW 90, NS 92, WJ 93, Te 93, Ch 93, Ch 94] have a number of deficiencies as discussed in Section 1.2.

1.1 Basic Concepts

In this section we introduce basic concepts in temporal databases. [Sn 94] provides a critical comparison of object-oriented temporal languages. [Ch 94] gives a survey of temporal query languages. [Ta 93] is a compilation of recent temporal models and languages. A glossary of concepts in temporal databases can be found in [Je 94]. A temporal database is a database which support some aspect of time excluding user-defined time. (Temporal databases are also referred to as time-oriented databases and historical databases) The currently supported times are valid time and transaction time. A database supporting both valid and transaction times is called *bitemporal* database. The valid time of an event or notion is the time when the notion is true in the modeled reality or when the event happens in the modeled reality. The transaction time of an object or event is the time when the fact is stored in the database. In the traditional sense, the transaction time cannot be changed after the current time, since all modifications (insertions/deletions) to data are always in the past and it is impossible to change the past. The transaction time is also referred to as registration time, and extrinsic time. The user-defined time is an un-interpreted data type of date and time. User defined time is similar to other domains (like integer, real) in that it has no special query language support. It is usually used for attributes such as birth date. A conventional database system usually support a few user defined time domains such as *datetime*, *date*, *time* etc.

A temporal event may be associated with any number of instants, intervals and temporal elements representing the valid time of the event. An instant is an isolated point in time. An interval is time between two instants. An interval may be represented by a set of contiguous instants. A temporal element is a finite set of intervals. for efficiency purposes a temporal element can be represented by a minimal number of maximal intervals in the set, i.e., a set of non-overlapping intervals. The lifespan of a database object or event is the time over which it is defined. The valid time lifespan of an event or object refers to the time when the corresponding object or event exists in the modeled reality. If the object has time stamp associated with it,

then the lifespan of the object is the value of its time stamp. In this context, an object might be a temporal (time-varying) attribute, tuple or object, or a relation or a class. A time stamp is the value associated with some temporal attribute, object, or event. This concept can be specialized further to valid-time stamp, transaction time stamp, bitemporal time stamp.

1.2 Previous Work

The aperiodic events in temporal databases can be modeled by aperiodic temporal databases such as [GV 85, T 86, CC 87, EW 90]. The periodic events, absolute or relative, however can not be modeled in these temporal models because these models do not support any periodic types or constructs for periodic events.

Intervals [All 83, All 84] and interval calculus are useful for modeling simple events with a beginning and an end time. A temporal element [GV 85] is a set of intervals used to represent aperiodic events happening over a finite set of intervals. Since periodic events happen indefinitely over infinite number of intervals, temporal element and its algebra can not be used for modeling periodic events either. Collection of intervals [L 86, Ca 94] is a temporal type similar to temporal elements in which intervals are hierarchically ordered. Collection of intervals are used to express various calendar expressions nicely in a calendar system such as the Gregorian Calendar System. Collection of intervals are restricted to modeling calendars and calendric expressions. Chandra et al [Ch 94] improved and implemented the calendar algebra in [L 86]. This implementation supports periodic events only within a finite interval of time. Kabanza et al. [KSW 90] introduced generalized tuples and generalized relations to model infinite periodic data. A generalized tuple consists of a set of attributes, two linear repeating points (lrp) and a set of constraints on these *lrp*s. A lrp is a finite set of equally spaced time instants denoted by a linear formula. Constraints are linear inequalities involving two lrps. A generalized relation is a set of generalized tuples. Although it is an interesting approach to model periodic events, it is not convenient because it is based on the

instant type. The use of *lrp*s gives rise to the duplication of data over multiple tuples. Niezette and Stevenne proposed in [NS 92] linear repeating intervals (*lri*) instead of *lrp*s to overcome the difficulties of the *lpr*s in expressing calendars within the same theoretical framework. A *lri* is a possibly infinite set of fixed-length equally-spaced set of intervals. However some difficulty with generalized databases still exists. For example, a calendric event or just one calendar may need to be stored as a set of generalized tuples resulting in data duplication, because a calendar may correspond to more than one *lri* [NS 92]. An object-oriented approach is taken for modeling calendars in Eiffel language in [Te 93]. The implementation of calendars are based on instants (time points), intervals, and duration types in a similar fashion to [So 93] where the representation independent and calendar specific properties of periodic events are handled separately.

Michael Soo [So 93] proposed an extension to SQL2 [Me 94] that supports multiple calendars. The main idea of this proposal is to separate the user dependent features of calendars from a universal ones. The SQL2 extension provides the set of universal features for calendars, and leaves the user dependent aspects to the database managers. Temporal mediators [WJ 93] are a means for converting between different calendar granularity. The main purpose of this approach is to display the data in a temporal relational database in terms of a different time granularity.

The approaches mentioned above have a number of deficiencies: First of all, none of the above approaches distinguishes between the *aperiodic* and *strictly-periodic* part of a *partially-periodic* event. Consider an employee who is employed part-time after being a temporary employee for 3 weeks. The work history of this employee is partially-periodic, i.e., the work-hours as a temporary employee is aperiodic, while the work-hours as a part-time employee is strictly-periodic. With the existing approaches, such a temporal event has to be stored as at least two separate objects resulting in (*i*) data duplication, (*ii*) inefficient query processing and (*iii*) difficulty in maintaining data integrity. Second, the reasoning about periodic time is not fully automated, the user has to write a significant amount of code to describe

how to do the conversions between time granularity as in [Ch 94]. Third, calendric events defined periodically in terms of two other events like *every year between Thanksgiving and Christmas* can't be expressed by the existing approaches at symbolic level. Fourth, a good periodic temporal algebra should support the *and, or,* and *not* of natural languages. To our knowledge there is not a periodic temporal type supporting all of the set theoretic operations. Fifth, the existing approaches are usually concerned with modeling periodic time in the context of calendars and neglect the issue of supporting periodic events in a temporal data model. Finally, the existing approaches do not attempt to model relative events which frequently occur in real world applications such as the temporal templates mentioned in Example 1.1.

1.3 Summary

Our main objective is to model periodic time by means of a new temporal type powerful enough that can be applied to a variety of application domains. The contribution of this thesis can be summarized as follows:

1. *Modeling periodic time in general through a new temporal type called periodic element*: By modeling periodic time independent of any application domain, we achieved an domain-independent formal framework which can be tailored to the specific needs of an application easily as mentioned in points 2, 3, and 4 below. We view periodic events as a generalization of aperiodic events, i.e., a periodic event is an aperiodic event happening periodically. Furthermore we allow a periodic event to be a partially-periodic event to capture the semantics of many real-world applications.
2. *Modeling relative time*: Event though there are many applications requiring relative time, the issue of representing and reasoning about relative time has not been addressed properly in the literature. In spite of the fact that there are many similarities between absolute time and relative time, they are not the same. The relative time support such as those by Tquel, TSQL2 deal with a limited form of aperiodic relative time. We propose relative aperiodic types *relative intervals,*

relative temporal element for modeling relative aperiodic events and *relative periodic element* for modeling relative partially-periodic events. The behavior of these types are again defined via the set theoretic operators, the temporal transformations, the temporal relationship operators with clear semantics.

3. *Extending the object-oriented model with periodic elements for developing, maintaining, and querying applications involving periodic events*. We used an object-oriented database management system to support periodic time by modeling temporal values as a function of time and periodic element as an abstract data type. This temporal extension is also implemented to demonstrate the feasibility of the basic ideas of this research. The first part of the implementation involves implementing the aperiodic types of instant, interval, and temporal element and the new periodic type of periodic element. In the second part, we implemented the periodic temporal object oriented model in O_2 database management system. Finally a set of temporal queries are executed against a sample database.
4. *Modeling calendars and calendric events using periodic elements*. We consider calendars as a special type of periodic events and express calendric expressions through the operators of periodic elements. Specifically introduced are the *periodic interval* and *periodic interval selection* operators to formulate complex calendric expressions.

The rest of the thesis is organized as follows: The aperiodic temporal types and operators are discussed in Chapter 2. Modeling periodic time as a periodic type is discussed in Chapter 3. The periodic object oriented data model is presented in Chapter 4. The implementation is studied in Chapter 5. Modeling calendars and calendric events is discussed in Chapter 6. Chapter 7 is reserved for the related work and comparison. Finally we summarize and conclude in Chapter 8.

Chapter 2

Background

In this chapter we discuss the aperiodic temporal types *instants*, *intervals* and *temporal elements*. Similar and more extensive discussions about these types can be found in the literature [All 83, Ga 88]. There are reasons for discussing these type here:

1. The new periodic type *periodic element* and its algebra introduced in the next chapter are formally based on the *temporal element* type*, which* in turn is based on the *interval* type, and so on.
2. The aperiodic time is also supported by our periodic temporal model introduced in Chapter 4. Therefore we need these types to support aperiodic temporal events.

We adhere to the following format in discussing the aperiodic types: First, the types are discussed in the order of increasing complexity, i.e., instant, intervals, and temporal elements, where the operations of a simpler type are given in terms of the operations of the preceding types. Second, each individual type is presented as follows: The type is defined first; then four classes of operators are defined on the type:

1. *Temporal transformations*: Two types of temporal transformations, namely *translation* and *scaling,* are defined for each type. *Translation* is used for describing the changes to the beginning time of an event such as a time delay. *Scaling* represents the changes to the duration of an event such as the fast forward operation on a audio stream in a multimedia application.
2. *Specialized operators*: These are operators defined specifically for the type. The *distance* (number of time units) between two temporal event, or the *length* (duration) of a simple temporal event can be counted among these.

3. *Set theoretic operators*: These operators (*union* ∪, *intersection* ∩, *and complementation* ~) corresponding to the *or*, *and*, and *not* connectives constitute an algebra for the type. They allow us to express complex temporal events in terms of simpler ones. Foe example the duration of an event may be given as the intersection of two other temporal events. Clearly a good temporal algebra should be closed under the set theoretic operators.

4. *Relational operators*: The temporal relationships between two temporal events can be captured by the temporal relationship operators *before*, *after*, *equal*, *overlap*, etc. These operators return Boolean values and are used in expressing temporal queries in temporal calculus such as the one in [All 83]. Relational operators for instants and intervals are defined in a straightforward way and understood well.

Third, we will use double-underlines, single-underline, and bold typeface whenever necessary to distinguish between the operators of intervals, temporal elements, and instants. The operators of periodic elements in the next chapter will be written without any discriminatory marks. In the rest of the thesis, we refer to *temporal element* as simply *element* for brevity. Fourth, the presentation method described here will also be used in the next chapter to define periodic elements.

The following notations are used in the rest of the thesis: The instants are represented by lowercase letters. The uppercase letters I, J, K, L, ... usually stand for intervals. The uppercase letters also depict elements. The Greek letters α, β, γ, ... represent periodic elements. 0v!6v!+v!'v!2 stand for the natural numbers, the set of all instants (the time-line), intervals, elements and periodic elements respectively. *now* denotes the current time point and ∅ stands for the empty element. ∞ is the positive infinity. For the sake of simplicity we assume that time is discrete, i.e., 6!µ!0. However, it is possible to represent time with real numbers (*continuous time*).

The implementation of the types discussed in this section along with their operators are given in O_2C in Appendix. We point out in Chapter 5 on how to implement other types of time with minimal changes to the *instant* and *interval* types.

2.1 Instant

Definition 2.1 (Instant)

An instant $t \in 6$ *is a point in time representing an instantaneous event.*

□

A. Temporal transformations: Let d, s $\in$ 0 and t be an instant. Then

- translate (t, d) = t + d.
- scale (t, s) is undefined.

B. Specialized operators: Let i, j $\in$ 6. Then

- previous (i) = i – 1, for i ≥ 1.
- next (i) = i +1, for i ≥ 0.
- distance (i, j) = absolute (j – i).
- max (i, j) =

  i   if i **after** j = true,

  j   otherwise.
- min (i, j) =

  i   if i **before** j = true,

  j   otherwise.

C. Relational operators: There are 3 basic relationships between given two time points t_1 and t_2:

- t_1 **before** t_2 = true ⇔ $t_1 < t_2$.
- t_1 **after** t_2 = true ⇔ $t_1 > t_2$.
- t_1 **equals** t_2 = true ⇔ $t_1 = t_2$.

Other relationships such as before-or-equal can be defined:

(t_1 **beforeorequal** t_2 = true) ⇔ (t_1 before t_2 = true) or (t_1 equal t_2 = true).

D. Set theoretic operators: The set theoretic operators are undefined for instants.

2.2 Interval

Definition 2.2 *(Interval)*

An interval I ∈ +denoted by [b, e] is a continuos segment of time-line T:

*[b, e] = {t | t ∈ T, b **beforeorequal** e = true, b **beforeorequal** t = true, t **beforeorequal** e = true}*

where b and e are called the beginning and end of interval I respectively.

□

For example, *[4, 7]* represents the time between instants 4 and 7 inclusive. The open ended intervals *(b, e], [b, e),* and *(b, e)* are defined similarly. *t* is a shorthand for *[t, t]*.

A. Temporal transformations: Let [t_1, t_2] be an interval and d, s ∈ 0. Then

- translate ([t_1, t_2], d) = [**translate** (t_1, d), **translate** (t_2, d)].
- scale ([t_1, t_2], s) = [t_1, **translate** (t_1, **distance** (t_1, t_2)s].

B. Specialized operators:

- begin ([t_1, t_2]) = t_1.
- end ([t_1, t_2]) = t_2.
- length (I) = **distance** (begin (I), end (I)).
- distance (I, J) =

distance (begin (I), end (J))	if J before I = true,
distance (begin (J), end (I))	if I before J = true,
0	otherwise.

The before operator is defined in Table 2.1.

Table 2.1: Relational operators on intervals.

Operator	Semantics
I before J = true ⇔	end (I) **before** begin (J) = true
I after J = true ⇔	begin (I) **after** end (J) = true
I meets J = true ⇔	end (I) **equal** begin (J) = true
I metby J = true ⇔	end (J) **equal** begin (I) = true
I leftoverlaps J = true ⇔	begin (I) **before** begin (J) = true & end (I) **after** begin (J) = true & end (I) **before** end (J) = true
I rightoverlaps J = true ⇔	begin (I) **after** begin (J) = true & begin (I) **before** end (J) = true & end (I) **after** end (J) = true
I contains J = true ⇔	begin (I) **beforeorequal** begin (J) & end (J) **beforeorequal** end (I)
I containedby J = true ⇔	begin (J) **beforeorequal** begin (I) = true & end (I) **beforeorequal** end (J) = true
I starts J = true ⇔	begin (I) **equal** begin (J) = true & end (I) **before** end (J) = true
I startedby J = true ⇔	begin (I) **equal** begin (J) = true & end (I) **after** end (J) = true
I finishes J = true ⇔	end (I) **equal** end (J) = true & begin (I) **before** begin (J) = true
I finishedby J = true ⇔	end (I) **equal** end (J) = true & begin (I) **after** begin (J) = true
I equals J = true ⇔	begin (I) **equal** begin (J) = true & end (I) **equal** end (J) = true

C. Relational Operators: Given 2 intervals I and J, there can be 13 possible temporal relationships between the two intervals [All 83] as given in Table 2. These relationships are illustrated in Figure 2.1. It is also possible to define new operators from the existing ones. Consider the following for an example:

$$(\text{I adjacent J} = \text{true}) \Leftrightarrow (\text{I meet J} = \text{true}) \text{ or } (\text{I metby J} = \text{true}).$$

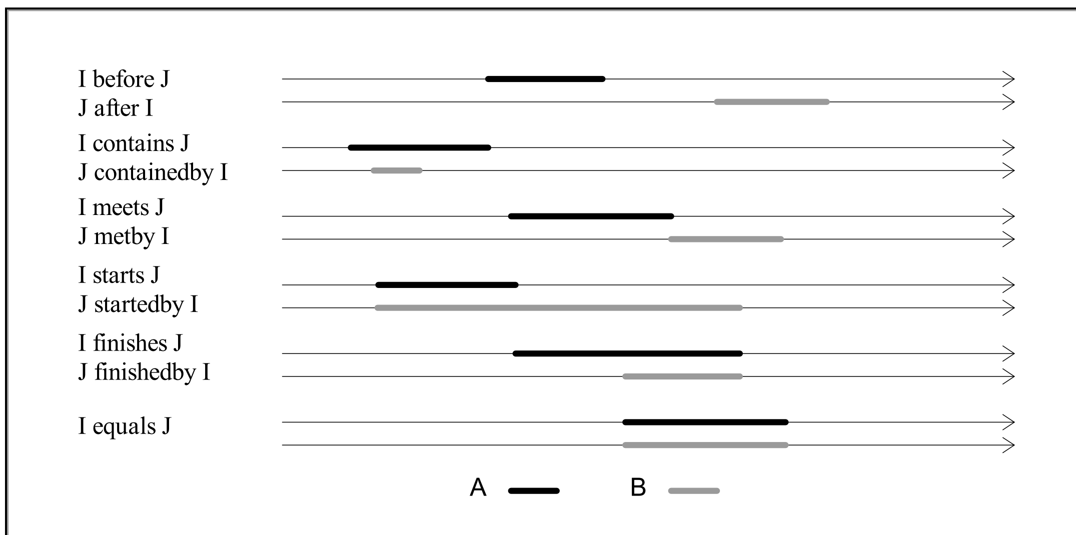

Figure 2.1: Relational Operators.

D. Set theoretic operators: These operators are defined in Table 2.2. The set of intervals are not closed under the set theoretic operators, because the intersection and complementation of two intervals can be a set of intervals [Al 83].

Table 2.2: The set theoretical operators on intervals.

Operator		*Semantics*	
$[t_1, t_2] \underline{\underline{\cup}} [t_3, t_4]$	=	$[\min(t_1, t_3), \max(t_2, t_4)]$	if $[t_1, t_2]$ overlap $[t_3, t_4]$ = true,
		$[t_1, t_2]$ and $[t_3, t_4]$	otherwise result is two intervals.
$[t_1, t_2] \underline{\underline{\cap}} [t_3, t_4]$	=	$[\max(t_1, t_3), \min(t_2, t_4)]$	if $[t_1, t_2]$ overlap $[t_3, t_4]$ = true
		$\varnothing$	otherwise.
$\underline{\underline{\sim}} [t_1, t_2]$	=	$[0, t_1]$ and $[t_2, \infty)$	if $t_1 \neq 0$,
		$[t_2, \infty)$	if $t_1 = 0$.

2.3 Temporal Element

Definition 2.3 (Temporal element)

A temporal element $S \in 5$ is a finite set of intervals $I_1, I_2, \ldots, I_n$ representing an event happening over more than one interval of time [GV 85]

$$S = (I_1, I_2, \ldots, I_n) = I_1 \underline{\underline{\cup}} I_2 \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} I_n$$

where $n \geq 0$ is an integer.

□

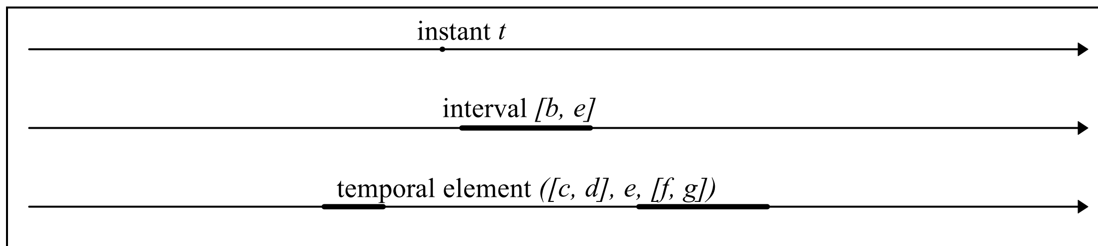

Figure 2.2: Aperiodic types and Periodic Element.

Parenthesis can be omitted when there is no confusion. By definition every interval *I* is a temporal element denoted by the interval itself. A temporal element such as (*[3, 7], [5, 10])* may be specified inefficiently because of the overlapping intervals.

Canonical form for temporal elements is defined to eliminate redundancies in the representation.

Definition 2.4 (Canonical temporal element)

Given a temporal element S, S is called canonical temporal element if the intervals I_1, I_2, ..., I_n *are pair wise-disjoint:*

$$\forall I, J\ ((I, J \in S)\ \&\ (I\ \underline{equal}\ J = false) \Rightarrow I\ \underline{overlaps}\ J = false).$$

□

The operators *overlaps and equal* are defined in Table 2.3. The corresponding canonical temporal element for *([3, 7], [5, 10])* is *[3, 10]*.

Definition 2.5

Given an arbitrary temporal element S, the function canonical : 5 → 5 returns the canonical form of S:

$$canonical\ (S) = Q,$$

where Q is the canonical form of S.

□

The procedural definition of *canonical* can be found in Appendix 9.5.1.

A. Temporal transformations: Let $(I_1, I_2, ..., I_n)$ be a temporal element. Then

- $translate\ ((I_1, I_2, ..., I_n), d)$ = $(\underline{translate}\ (I_1), \underline{translate}\ (I_n), ..., \underline{translate}\ (I_n))$.
- $scale\ ((I_1, I_2, ..., I_n), s)$ = $(\underline{translate}\ (\underline{scale}\ (I_1, s), d_1)),$
  $\underline{translate}\ (\underline{scale}\ (I_2, s), d_2)),$
  ...,
  $\underline{translate}\ (\underline{scale}\ (I_n, s), d_n)))$

where d_k = **distance** (begin (I_k), begin (I_1)) for $1 \leq k \leq n$.

B. Specialized Operators: Let $(I_1, I_2, ..., I_n)$ be a temporal element as before. Then

- first_interval $((I_1, I_2, ..., I_n)) = I_1$.
- select_interval $((I_1, I_2, ..., I_n), k) = I_k$ for $1 \leq k \leq n$.
- last_interval $((I_1, I_2, ..., I_n)) = I_n$.
- begin $((I_1, I_2, ..., I_n))$ = begin (I_1).
- end $((I_1, I_2, ..., I_n))$ = end (I_n).
- length $((I_1, I_2, ..., I_n))$ = length (I_1) + length (I_2) + ... + length (I_n).
- distance (S, Q) =

  distance (last_interval (S), first_interval (Q)) if S before Q = true,

  distance (last_interval (Q), first_interval (S)) if Q before S = true,

  0 otherwise.

C. Relational operators: Let A and B two element. Then 13 the temporal relationship operators on intervals are gracefully extended to elements as defined in Table 2.3. These operators are illustrated in Figure 2.3. The operators of temporal element

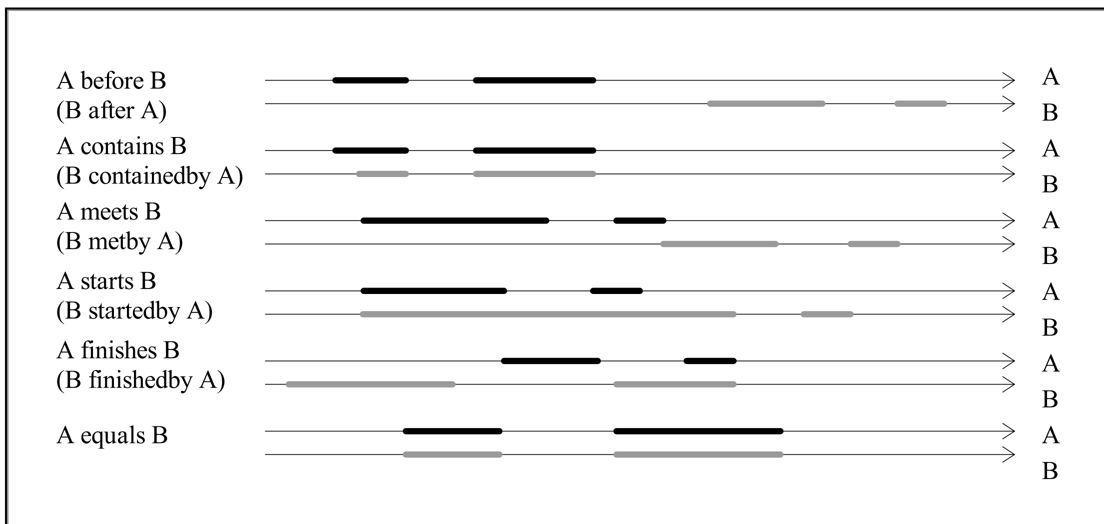

Figure 2.3: Relational operators on elements.

reduce to the operators of intervals, when operands are simple intervals instead of a set of interval, e.g.,

A before B = I before J $\Leftarrow$ (A = J) & (B = I).

Table 2.3: Relational operators on elements.

Operator				Semantics
A	before	B	= true ⇔	end (A) **before** begin (B) = true
A	after	B	= true ⇔	begin (A) **after** end (B) = true
A	meets	B	= true ⇔	end (A) **equal** begin (B) = true
A	metby	B	= true ⇔	end (B) **equal** begin (A) = true
A	leftoverlaps	B	= true ⇔	begin (A) **before** begin (B) = true & end (A) **after** begin (B) = true & end (A) **before** end (B) = true
A	rightoverlaps	B	= true ⇔	begin (A) **after** begin (B) = true & begin (A) **before** end (B) = true & end (A) **after** end (B) = true
A	contains	B	= true ⇔	begin (A) **beforeorequal** begin (B) & end (B) **beforeoreuqal** end (A)
A	containedby	B	= true ⇔	begin (B) **beforeorequal** begin (A) = true & end (A) **beforeoreuqal** end (B) = true
A	starts	B	= true ⇔	begin (A) **equal** begin (B) = true & end (A) **before** end (B) = true
A	startedby	B	= true ⇔	begin (A) **equal** begin (B) = true & end (A) **after** end (B) = true
A	finishes	B	= true ⇔	end (A) **equal** end (B) = true & begin (A) **before** begin (B) = true
A	finishedby	B	= true ⇔	end (A) **equal** end (B) = true & begin (A) **after** begin (B) = true
A	equals	B	= true ⇔	canonical (A) **=** canonical (B)

D. Set theoretic operators: The set theoretic operators union ($\underline{\cup}$), intersection ($\underline{\cap}$) and complementation ($\underline{\sim}$) are depicted in Figure 2.4. Let $A = (I_1, I_2, ..., I_k)$, $B = (J_1, J_2, ..., J_m)$ be canonical elements, and $I_i = [t_{2i-1}, t_{2i}]$. Then

- $A \underline{\cup} B$ = canonical $(I_1 \underline{\cup} I_2 \underline{\cup} ... \underline{\cup} I_k \underline{\cup} J_1 \underline{\cup} J_2 \underline{\cup} ... \underline{\cup} J_m)$.
- $A \underline{\cap} B$ = canonical $((I_1 \underline{\cap} J_1) \underline{\cup} (I_1 \underline{\cap} J_2) \underline{\cup} ... \underline{\cup} (I_k \underline{\cap} J_m))$.
- $\underline{\sim} A$ = canonical $([0, t_1] \underline{\cup} [t_2, t_3] \underline{\cup} ... \underline{\cup} [t_{2k}, \infty))$.

Theorem 2.1

The set of temporal elements are closed under the set theoretic operations of union, intersection, and complementation with T (the set of time instants) as its maximum element and ∅ as its minimum element and it forms a Boolean algebra [GV 85].

□

Temporal elements has been used as time stamps in a number of temporal data models including [G 88, T 93], because it can represent events happening over more

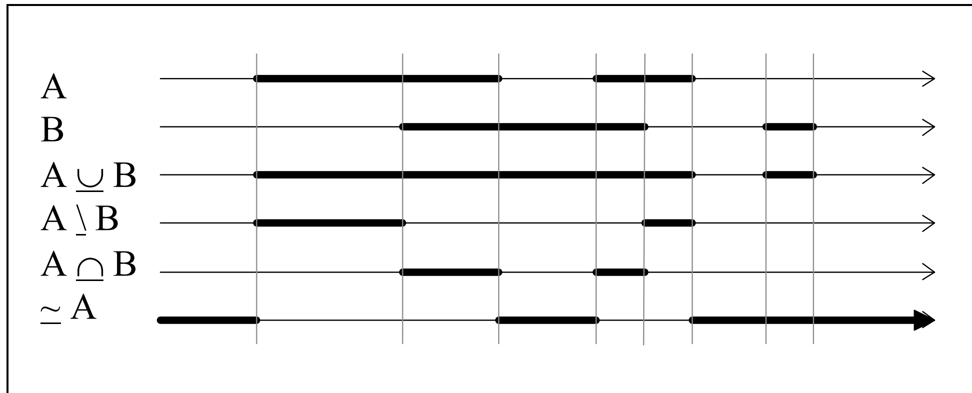

Figure 2.4: The Set theoretic operators on temporal elements.

than interval of time and is closed under the set theoretic operators.

2.4 Summary

The aperiodic temporal types instant (point), interval and temporal element are used in modeling time. An instant is a point in time representing an instantaneous event. An interval is a convex duration of time specified by its start and end times. Intervals denote events happening in a continuous segment of time. A temporal element is a finite set of intervals. A temporal event represents an event possibly happening over more than one segment of time. We studied the behavior of aperiodic time by identifying four basic sets of operators though which other more complex behavior can be defined: Temporal transformations, temporal relationship operators, specialized operators, and set theoretic operators.

Chapter 3

Modeling Periodic Time

We introduce a new temporal type called *periodic element* capable of representing a number of different types of periodic events in this Chapter. The differences between *periodic element* and the rest of the proposals on periodic time can be summarized as follows:

1. In the most general case, a periodic event consists of an *aperiodic part* and a *periodic part* called *partially-periodic event*. As an event evolves in time, the aperiodic and periodic part of the event may change which may cause an aperiodic event to become periodic, or a periodic event to become aperiodic event.

2. We consider aperiodic time a special case of periodic time, that is, periodic time is a generalization of periodic time, and partially-periodic time is a generalization of periodic time. Thus periodic element is backward compatible with aperiodic types temporal elements, intervals, and instants. The total syntactic and semantic backward compatibility for the above aperiodic types is provided.

3. The behavior of partially-periodic time is defined in a simple and user-friendly manner. The users of aperiodic time are already familiar with *(i)* temporal transformations, *(ii)* temporal relationship operators, *(iii)* set theoretic operators and some other already existing operators such as the *distance* between two events, the *length* of an event, etc. which we call *(iv)* specialized operators. Instead of defining completely new and different operators, we extend the definitions of these existing operators to partially-periodic events. This approach has two advantages: *(i)* Users familiar with aperiodic time can write queries for partially-periodic time with ease. There is minimal training required for such

users. *(ii)* the existing queries for aperiodic time don't need to change in the case of upgrading an aperiodic temporal system/database to partially-periodic time.

4. We pay special attention to representing and reasoning about relative events too. The types of relative interval, relative temporal element and relative periodic elements are defined for relative time. The behavior of relative time is defined similar to absolute time through temporal transformations, set theoretic operators, temporal relationship operators, etc.

5. Finally a generalized definition of partially-periodic time is given in which the aperiodic part consists of a sequence of *finite periodic events*. Finite periodic time provides compact and efficient representation of periodic events.

This chapter is organized as follows: Periodic element is introduced in Section 3.1. Relative periodic time and its properties are presented in Section 3.2. Finite periodic time is studied in Section 0. The chapter is concluded with a summary in Section **Error! Reference source not found.**.

3.1 Periodic Element as a Temporal Type

A *periodic element* consists of three components: (i) *aperiodic part* is a temporal element denoting the initial irregular temporal pattern of a temporal event, (ii) p*eriodic part* is also a temporal element denoting the regular indefinitely-many-repeating temporal pattern of the temporal event, and (iii) *period* is a duration of time denoting the distance between two consecutive occurrence of the *periodic part*.

Definition 3.1 (Periodic element)

A triplet of 2 temporal elements, aperiodic part N, periodic part P, and a duration (period) p $\in$ *0, is called a periodic element (denoted by N:P{p})*

$$N{:}P\{p\} = N \,\underline{\cup}\, P\{p\}$$

where

- $P\{p\} = P \,\underline{\cup}\, \underline{translate}(P, p) \,\underline{\cup}\, \underline{translate}(P, 2p) \,\underline{\cup} \ldots$

- *N and P don't not contain ∞,*
- *$p \geq 0$,*
- *N <u>before</u> P = true,*
- *p ≥ **distance** (<u>end</u> (P), <u>begin</u> (P)).*

□

A periodic element is essentially an infinite union of intervals with the periodic part repeating indefinitely as shown in Figure 3.1.

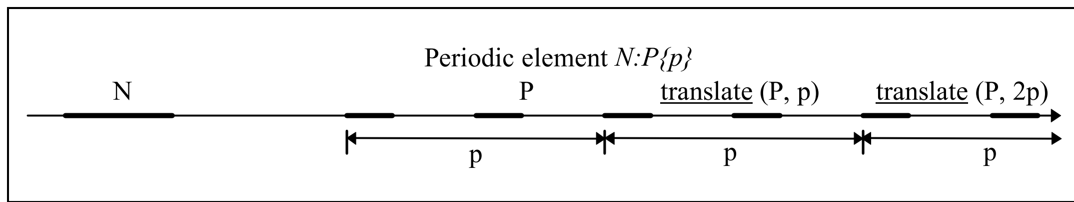

Figure 3.1: Periodic element.

The condition $p \geq 0$ ensures that the periodic event being represented extends into indefinite future. The condition *N <u>before</u> P = true* states that the aperiodic part of the event must happen *before* the periodic part to distinguish between the two. The fourth condition asserts that the period must be at least as long as the length of the periodic part of the event.

If the periodic part *P* of a periodic element *N:P{p}* is an empty element ($P = \varnothing$) or the period is zero ($p = 0$), then this periodic element reduces to temporal element *N* and the periodic part is omitted in representation, i.e., *N:∅{0}* and *N* are equivalent. If the aperiodic part *N* is an empty element ($N = \varnothing$), then the aperiodic part is omitted from the representation and *N:P{p}* is written *P{p}* for short, i.e., *∅:P{p}* and *P{p}* are equivalent.

3.1.1 Temporal transformations

The temporal transformations of periodic elements is a direct extension of the transformation of temporal elements. The translation shifts the whole event to the

left/right on the time axis, while the length of individual intervals remains the same. The scaling keeps the beginning time of the whole event the same, while stretching or shrinking the distance between any two time instants in the event. Temporal transformations are illustrated in Figure 3.2.

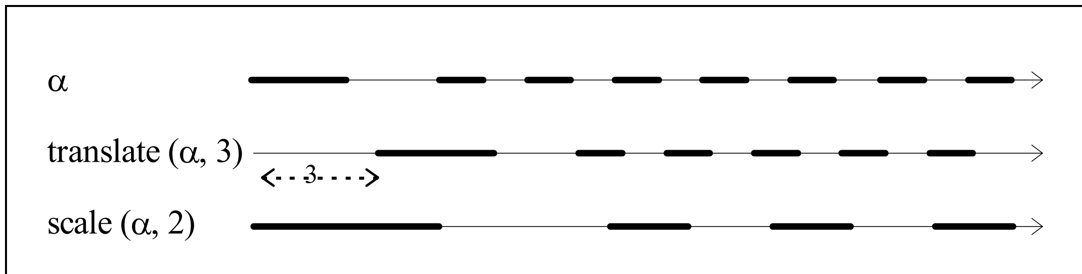

Figure 3.2: Temporal transformations of periodic elements

- translate (N:P{p}, d) = <u>translate</u> (N, d) : <u>translate</u> (P, d){p}.
- scale (N:P{p}, s) = <u>scale</u> (N, s) : <u>translate</u> (<u>scale</u> (P, s), <u>begin</u>(P)s){ps}.

3.1.2 Specialized Operators:

Let α, β and N:P{p} be periodic elements, i an integer. Then

- begin (N:P{p}) =
  - <u>begin</u> (N) if N ≠ ∅,
  - <u>begin</u> (P) otherwise.

- end (N:P{p}) =
  - <u>end</u> (N) if P = ∅,
  - ∞ otherwise.

- first_interval (N:P{p}) =
  - <u>first_interval</u> (N) if N ≠ ∅,
  - <u>first_interval</u> (P) otherwise.

- last_interval (N:P{p}) =
  - <u>last_interval</u> (N) if P = ∅.

- select_interval (N:P{p}, i) =

select_interval (N, i) if i ≤ # of interval (N),
select interval (P, (i – # of interval (N)) mod # of interval (P)) otherwise.

- length (N:P{p}) =
  length (N) if P = ∅,
  ∞ otherwise.

- distance (α, β) =
  distance (begin (α), end (β)) if begin (β) **before** end (α) = true,
  distance (begin (β), end (α)) if begin (α) **before** end (β) = true,
  0 otherwise.

Definition 3.2

Given a periodic element α = N:P{p}, the functions aperiodic_part: 2 → ', periodic_part: 2 → ' and period: 2 → 0 returns, the aperiodic part N, the periodic part P{p}, and the period p of N:P{p} respectively:

aperiodic_part (N:P{p}) = N.

periodic_part (N:P{p}) = P.

period (N:P{p}) = p.

□

The representation of periodic events is important for time and space efficiency. Redundancy in representation results in poor time efficiency. For example, the complexity of the set theoretic operations or the formatting functions defined below is given in terms of the number of intervals in the periodic event. Therefore we define an efficient representation for periodic elements, called canonical periodic elements, based on canonical elements.

Definition 3.3

A periodic element N:P{p} is said to be canonical if

- *N and P are canonical elements,*

- *N* $\underline{\text{meets}}$ *P = false.*

□

The first condition ensures that the representation for aperiodic and periodic parts are not redundant. The second condition ensures that an interval is not divided between the aperiodic and periodic part of a periodic element. Let *[0, 3]* $\underline{\cup}$ *[2, 7] : (7, 10] {5}* be a periodic element. The corresponding canonical periodic element is given as *[0, 10] : (12, 15] {5}*. The canonical form of a periodic element can be computed as follows:

Procedure: Computes the canonical form for a given periodic element.

Input: Periodic element N:P{p].

Output: The canonical form of N:P{p}.

Method:

1. N ← $\underline{\text{canonical}}$ (N).
2. P ← $\underline{\text{canonical}}$ (P).
3. If *N* $\underline{\text{meets}}$ *P = True*
   N ← N $\underline{\cup}$ first_interval (P),
   P ← (P $\underline{-}$ first_interval (P)) $\underline{\cup}$ ($\underline{\text{translate}}$ (first_interval (P), p)).
4. Simplify periodic part: *(assume P* = (I_1, ..., I_n)*)*
   for *i* = $\lfloor n/2 \rfloor$ *downto 2*
   if unfold ((I_1, ..., I_i), n/i) equal P
   N ← (I_1, ..., I_i), exit loop.
5. Simplify aperiodic part: (*assume N* = (I_1, ..., I_k))
   for I = k down to 1
   if (I_1, ..., I_k):(I_1, ..., I_n){p} equal (I_1, ..., I_{k-1}):(I_k, I_1..., I_{n-1}){p}
   N:P{P} ← (I_1, ..., I_{k-1}):(I_k, I_1..., I_{n-1}){p}.
6. Return N:P{p}.

In Step 1 and 2, the aperiodic and periodic part of the periodic element are transformed in to canonical form (the *canonical* function is defined in Section 2.3). In Step 3, if an interval spans over both aperiodic and periodic part, then it is combined into aperiodic part and the periodic part is adjusted accordingly. Step 4 simplifies the

periodic part so that in can not be folded anymore. Step 5 simplifies the aperiodic part so the periodic element can not be left-shifted anymore.

3.1.3 Formatting Functions

In order to defined the set theoretic operators and canonical form for periodic elements, we first define a set of functions, called *formatting* functions, on periodic elements. These functions (called *right, left, fold, unfold*), in principal, rewrite a periodic element. The former two shift the beginning of the periodic part of a periodic event to the right/left, while the latter two fold/unfold the periodic part of a periodic event. Thus these operators can be used to rewrite a periodic event in a number of ways. Let us illustrate these function with an example first.

Example 3.1

Refer to Figure 3.3 for this example. Let α *= [0, 4], [5, 6] : [7, 8],[9, 10]{4}. Then*

- *left (*α*, 2) = [0, 4] : [5, 6], [7, 8]{4}*
- *right (*α*, 2) = [0, 4], [5, 6], [7, 8] : [9, 10], [11, 12]{4}*
- *fold (*α*, 2) = [0, 4], [5, 6] : [7, 8]{2}*
- *unfold (*α*, 2) = [0, 4], [5, 6] : [7, 8], [9, 10], [11, 12], [13, 14]{8}.*

□

The formatting functions should not be confused with the temporal transformations

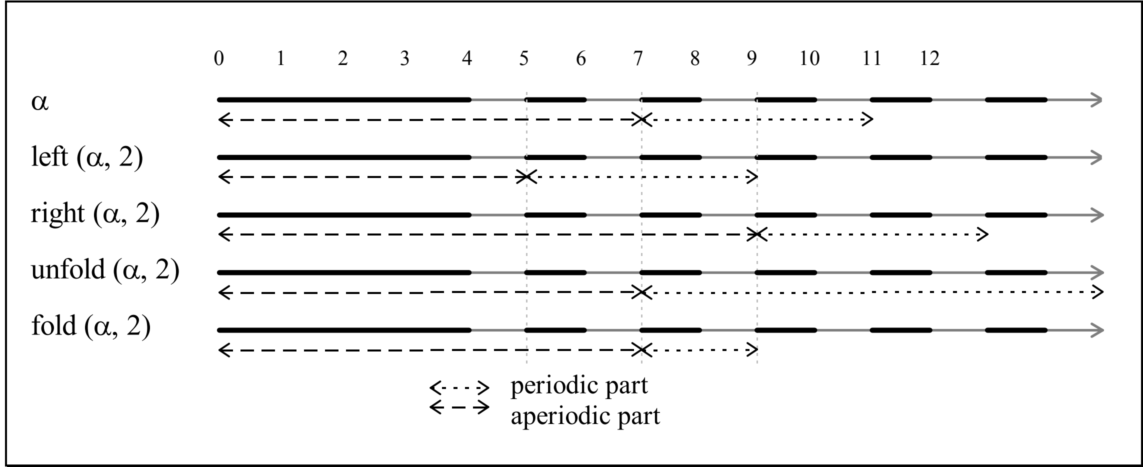

Figure 3.3: Formatting functions on periodic elements

even though there is similarity between *translation* and *right/left* functions and between *scaling* and *fold/unfold* functions. The result of a *transformation* of a periodic event is a different event, while the result of *formatting* is the same event expressed differently. We now formally describe these formatting functions.

Definition 3.4 (Right-shift)

Given a periodic element N:P{p} and a duration d ≥ 0, the function right : 2 ҡ !0 → 2 right-shifts the beginning of the periodic part of α by d time units:

$$right\ (N{:}P\{p\}, t) = N \ \underline{\cup}\ \Big(\bigcup_{i=1}^{k} \underline{translate}\ (P, (i-1)p)\Big)\ \underline{\cup}\ (\underline{translate}\ (P, kp)\ \underline{\cap}\ [\underline{begin}\ \underline{translate}\ (P, kp)), \underline{translate}\ (\underline{begin}\ (P), d)]) : [\underline{translate}\ (\underline{begin}\ (P), d), \underline{translate}\ (\underline{begin}\ (P), d+p)]\ \underline{\cap}\ (\underline{translate}\ (P, kp)\ \underline{\cup}\ \underline{translate}\ (P, (k+1)p))\ \{p\}$$

where $k = \lfloor d / p \rfloor$.

□

Definition 3.5 (Left-shift)

Given a periodic element N:P{p} and a duration d ($0 \leq d \leq p$)*, the function left: 2 !x !0 → 2 shifts the beginning of the periodic part P of N:P{p} by d time units to the left. Let* $Q = N \underline{\cap} [\underline{translate}\ (\underline{begin}\ (P), -d), \underline{begin}\ (P)]$*. Then*

$$left\ (N{:}P\{p\}, d) = N \underline{-} Q : Q \underline{\cup} (P \underline{\cap} [\underline{begin}\ (P), \underline{translate}\ (\underline{begin}\ (P), p-n)])\{p\}$$

if $Q = \underline{translate}\ ((P \underline{\cap} [\underline{translate}\ (\underline{begin}\ (P), p\text{-}n), \underline{end}\ (P)]), p)$.

□

Definition 3.6 (Fold)

Given a periodic element N:P{p} and an integer $n > 0$*, the function fold: 2 !x !0 → 2 divides the periodic part of* α *into q equal-length, identical-pattern temporal elements as follows:*

$$fold\ (N{:}P\{p\}, q) = N : [\underline{begin}\ (P), \underline{translate}\ (\underline{begin}\ (P), p/n)] \underline{\cap} P\ \{p/n\}$$

if $\underline{translate}\ (Q_i, (j-i)p/n) = Q_j$ *for* $1 \leq i < j \leq n$ *where*

$$Q_k = [\underline{translate}\ (\underline{begin}\ (P), (k-1)p/n), \underline{translate}\ (\underline{begin}\ (P), kp/q)] \underline{\cap}\ P.$$

□

Definition 3.7 *(Unfold)*

Given a periodic element N:P{p} and an integer n (> 0)*, the function unfold replicates the periodic part of* α *n times as follows:*

$$unfold\ (N{:}P\{p\}, n) = N : \bigcup_{i=1}^{n} \underline{translate}\ (P, (i-1)p)\{pq\}.$$

□

The formatting functions are used to write a periodic event in different ways. For instance, the formatting functions *left* and *fold* can be used to eliminate the

redundancy in representation as shown in the above example. They do not change the periodic event being represented, i.e.,

$$\alpha = left\ (\alpha, 2) = right\ (\alpha, 2) = fold\ (\alpha, 2) = unfold\ (\alpha, 2).$$

The following lemmas state that *left* (α, d), *right* (α, d), *fold* (α, i) *and unfold* (α, i) can be used interchangebly for α in an expression for valid values of d and i.

Lemma 3.1 (Right Shift Rule)

Given a periodic element $N{:}P\{p\}$ *and a integer* $d \geq 0$

$$right\ (N{:}P\{p\}, d) = N{:}P\{p\}.$$

□

Proof

Let $k = \lfloor n / p \rfloor$. By the definition of periodic elements

$$N{:}P\{p\} = N \cup P \cup \underline{translate}\ (P, p) \cup \underline{translate}\ (P, 2p) \cup \ldots \cup \underline{translate}\ (P, kp) \cup \underline{translate}\ (P, (k+1)p) \cup \ldots$$

Let $\underline{translate}\ (P, kp) = Q_1 \cup Q_2$ *where*

$$Q_1 = \underline{translate}\ (P, kp) \cap [\underline{begin}\ (\underline{translate}\ (P, kp)), \underline{translate}\ (\underline{begin}\ (P), d)]$$

$$Q_2 = \underline{translate}\ (P, kp) \cap [\underline{translate}\ (\underline{begin}\ (P), d), \underline{translate}\ (\underline{begin}\ (\underline{translate}\ (P, kp), p)]$$

Then

$$N{:}P\{p\} = N \cup P \cup \underline{translate}\ (P, p) \cup \ldots \cup \underline{translate}\ (P, kp) \cup (Q_1 \cup Q_2) \cup \underline{translate}\ ((Q_1 \cup Q_2), p) \cup \underline{translate}\ ((Q_1 \cup Q_2), 2p) \cup \ldots$$

Distribute *translate* over $\cup$:

$$N{:}P\{p\} = N \underline{\cup} P \underline{\cup} \underline{translate}\ (P, p) \underline{\cup} \ldots \underline{\cup} \underline{translate}\ (P, kp) \underline{\cup} Q_1 \underline{\cup} Q_2 \underline{\cup} \underline{translate}\ (Q_1, p) \underline{\cup} \underline{translate}\ (Q_2, p) \underline{\cup} \underline{translate}\ (Q_1, 2p) \underline{\cup} \underline{translate}\ (Q_2, 2p) \underline{\cup} \ldots$$

Substitute N_1 *for* $N \underline{\cup} P \underline{\cup} \underline{translate}\ (P, p) \underline{\cup} \underline{translate}\ (P, 2p) \underline{\cup} \ldots \underline{\cup} \underline{translate}\ (P, (k-1)p) \underline{\cup} Q_1$ and P_1 *for* $Q_2 \underline{\cup} translate\ (Q_1, p)$:

$$N{:}P\{p\} = N_1 \underline{\cup} P_1 \underline{\cup} \underline{translate}\ (P_1, p) \underline{\cup} \underline{translate}\ (P_1, 2p) \ldots$$

From the definition of periodic element

$$N{:}P\{p\} = N_1{:}P_1\{p\}$$

which gives the following formula when the variables are substituted back

$$N{:}P\{p\} = N \underline{\cup} P \underline{\cup} \underline{translate}\ (P, p) \underline{\cup} \underline{translate}\ (P, 2p) \underline{\cup} \ldots \underline{\cup} \underline{translate}\ (P, (k-1)p) \underline{\cup} Q_1 : Q_2 \underline{\cup} translate\ (Q_1, p)\ \{p\}$$

The term $Q_2 \underline{\cup} translate\ (Q_1, p)$ reduces to $[\underline{translate}\ (\underline{begin}\ (P), d), \underline{translate}\ (\underline{begin}\ (P), d+p)] \underline{\cap} (\underline{translate}\ (P, kp) \underline{\cup} \underline{translate}\ (P, (k+1)p))$:

$$N{:}P\{p\} = N \underline{\cup} P \underline{\cup} \underline{translate}\ (P, p) \underline{\cup} \underline{translate}\ (P, 2p) \underline{\cup} \ldots \underline{\cup} \underline{translate}\ (P, (k-1)p) \underline{\cup} (\underline{translate}\ (P, kp) \underline{\cap} [\underline{begin}\ (\underline{translate}\ (P, kp)), \underline{translate}\ (\underline{begin}\ (P), d)]) : [\underline{translate}\ (\underline{begin}\ (P), d), \underline{translate}\ (\underline{begin}\ (P), d+p)] \underline{\cap} (\underline{translate}\ (P, kp) \underline{\cup} \underline{translate}\ (P, (k+1)p))\ \{p\}$$

From the definition of the function *right*

$$N{:}P\{p\} = right\ (N{:}P\{p\}, d).$$

□

Lemma 3.2 (Left Shift Rule)

Given a periodic element $N{:}P\{p\}$ *and an integer* d, $0 \le d \le p$

$$N{:}P\{p\} = left\ (N{:}P\{p\}, t)$$

if $N \cap [\underline{translate}\ (\underline{begin}\ (P), -d), \underline{begin}\ (P)] = \underline{translate}\ ((P \cap [\underline{translate}\ (\underline{begin}\ (P), p-n), \underline{end}\ (P)]), p)$.

□

Proof

Assume $N \cap [\underline{translate}\ (\underline{begin}\ (P), -d), \underline{begin}\ (P)] = \underline{translate}\ ((P \cap [\underline{translate}\ (\underline{begin}\ (P), p-n), \underline{end}\ (P)]), p)$.

By definition

$$N{:}P\{p\} = N \cup P \cup \underline{translate}\ (P, p) \cup \underline{translate}\ (P, 2p) \cup \ldots$$

Let $N = N_1 \cup N_2$ and $P = P_1 \cup P_2$ where

$$N_1 = [\underline{begin}\ (N), \underline{translate}\ (\underline{begin}\ (P), -d)] \cap N$$

$$N_2 = [\underline{translate}\ (\underline{begin}\ (P), -d), \underline{begin}\ (P)] \cap N.$$

$$P_1 = [\underline{begin}\ (P), \underline{translate}\ (\underline{begin}\ (P), p-d)] \cap P$$

$$P_2 = [\underline{translate}\ (\underline{begin}\ (P), p-d), \underline{translate}\ (\underline{begin}\ (P), p)] \cap P.$$

Then with substitution

$$N{:}P\{p\} = N_1 \cup N_2 \cup P_1 \cup P_2 \cup \underline{translate}\ ((P_1 \cup P_2), p) \cup \underline{translate}\ ((P_1 \cup P_2), 2p) \cup \ldots$$

Distribute *translate* over $\cup$

$$N{:}P\{p\} = N_1 \cup N_2 \cup P_1 \cup P_2 \cup \underline{translate}\ (P_1, p) \cup \underline{translate}\ (P_2, p) \cup \underline{translate}\ (P_1, 2p) \cup \underline{translate}\ (P_2, 2p) \cup \ldots$$

Since $\underline{translate}\ (N_2, p) = P_2$, replace $\underline{translate}\ (P_2, kp)$ with $\underline{translate}\ (N_2, (k-1)p)$ for $k \geq 1$

$$N{:}P\{p\} = N_1 \cup N_2 \cup P_1 \cup \underline{translate}\ (N_2, p) \cup \underline{translate}\ (P_1, p) \cup \underline{translate}\ (N_2, 2p) \cup \underline{translate}\ (P_1, 2p) \cup \underline{translate}\ (N_2, 3p) \cup \ldots$$

$$N{:}P\{p\} = N_1 \cup N_2 \cup P_1 \cup \underline{translate}\ ((N_2 \cup P_1), p) \cup \underline{translate}\ ((N_2 \cup P_1), 2p) \cup \underline{translate}\ ((N_2 \cup P_1), 3p) \cup \ldots$$

From the definition of periodic elements

$$N{:}P\{p\} = N_1 : (N_2 \cup P_1)\ \{p\}.$$

Note that $N_1 = N - N_2$ from the definition of difference ($-$). The resulting formula is

$$N{:}P\{p\} = N - N_2 : N_2 \cup (P \cap [\underline{begin}\ (P), \underline{translate}\ (\underline{begin}\ (P), p-n)])\{p\}$$

From the definition of *left*

$$N{:}P\{p\} = left\ (N{:}P\{p\}, d).$$

□

Lemma 3.3 (Folding Rule)

Given a periodic element $N{:}P\{p\}$ and a integer $n \geq 1$

$$N{:}P\{p\} = fold\ (N{:}P\{p\}, n)$$

if $\underline{translate}\ (Q_i, (j-i)p/n) = Q_j$ *for all* $1 \leq i < j \leq n$ *where*

$$Q_k = [\underline{translate}\ (\underline{begin}\ (P), (k-1)p/n), \underline{translate}\ (\underline{begin}\ (P), kp/n)] \cap P.$$

□

Proof

By definition

$$N{:}P\{p\} = N \cup P \cup \underline{translate}\ (P, p) \cup \underline{translate}\ (P, 2p) \cup \ldots$$

Let $Q_k = [\underline{translate}\ (\underline{begin}\ (P), (k-1)p/n), \underline{translate}\ (\underline{begin}\ (P), kp/n)] \underline{\cap} P$ *for* $1 \le k \le n$. Since $\underline{translate}\ (Q_i, (j-i)p/n) = Q_j$ *for all* $1 \le i < j \le n$, we have

$$P = Q_1 \underline{\cup} Q_2 \underline{\cup} \dots \underline{\cup} Q_n$$

Replace $Q_1 \underline{\cup} Q_2 \underline{\cup} \dots \underline{\cup} Q_n$ for P

$$N{:}P\{p\} = N \underline{\cup} Q_1 \underline{\cup} Q_2 \underline{\cup} \dots \underline{\cup} Q_n\ \underline{translate}\ ((Q_1 \underline{\cup} Q_2 \underline{\cup} \dots \underline{\cup} Q_n), p) \underline{\cup} \underline{translate}\ ((Q_1 \underline{\cup} Q_2 \underline{\cup} \dots \underline{\cup} Q_n), 2p) \underline{\cup} \dots$$

Distribute $\underline{translate}$ over $\underline{\cup}$

$$N{:}P\{p\} = N \underline{\cup} Q_1 \underline{\cup} Q_2 \underline{\cup} \dots \underline{\cup} Q_n \underline{\cup} \underline{translate}\ (Q_1, p) \underline{\cup} \underline{translate}\ (Q_2, p) \underline{\cup} \dots \underline{\cup} \underline{translate}\ (Q_n, p) \underline{\cup} \dots \underline{\cup} \underline{translate}\ (Q_1, 2p) \underline{\cup} \underline{translate}\ (Q_2, 2p) \underline{\cup} \dots \underline{\cup} \underline{translate}\ (Q_n, 2p) \underline{\cup} \dots$$

Replace all Q_k s with $\underline{translate}\ (Q_1, (k-1)p/n)$ for k > 1

$$N{:}P\{p\} = N \underline{\cup} Q_1 \underline{\cup} \underline{translate}\ (Q_1, p/n) \underline{\cup} \underline{translate}\ (Q_1, 2p/n) \underline{\cup} \dots$$

From the definition of periodic elements

$$N{:}P\{p\} = N : Q_1\ \{p/n\}$$

Write Q_1 explicitly

$$N{:}P\{p\} = N : [\underline{begin}\ (P), \underline{translate}\ (\underline{begin}\ (P), p/n)] \underline{\cap} P\ \{p/n\}$$

Hence

$$N{:}P\{p\} = fold\ (N{:}P\{p\}, n)$$

□

Lemma 3.4 *(Unfolding Rule)*

Given a periodic element N:P{p} and a integer $n \ge 1$,

$$N{:}P(p) = unfold\ (N{:}P\{p\}, n).$$

□

Proof

By definition

$$N{:}P\{p\} = N \,\underline{\cup}\, P \,\underline{\cup}\, \underline{translate}\ (P, p) \,\underline{\cup}\, \underline{translate}\ (P, 2p) \,\underline{\cup} \ldots$$

For some $n \geq 1$, rewrite $N{:}P\{p\}$ as

$$N{:}P\{p\} = N \,\underline{\cup}\, P \,\underline{\cup}\, \underline{translate}\ (P, p) \,\underline{\cup}\, \underline{translate}\ (P, 2p) \,\underline{\cup} \ldots\ \underline{translate}\ (P, pn) \,\underline{\cup}\, \underline{translate}\ (P, (n+1)p) \,\underline{\cup} \ldots$$

Substitute Q_i for $\underline{\bigcup}_{j=1}^{n} \underline{translate}\ (P, ((i-1)n+(j-1))p)$ for $i, n \geq 1$ and regroup the terms

$$N{:}P\{p\} = N \,\underline{\cup}\, Q_1 \,\underline{\cup}\, \underline{translate}\ (Q_1, pn) \,\underline{\cup}\, \underline{translate}\ (Q_1, 2pn) \,\underline{\cup} \ldots$$

From the definition of periodic element

$$N{:}P\{p\} = N \,\underline{\cup}\, Q_1\ \{pn\}$$

Substitute $\underline{\bigcup}_{j=1}^{n} \underline{translate}\ (P, ((i-1)n+(j-1))p)$ for Q_i for $i, n \geq 1$

$$N{:}P\{p\} = N{:}P \,\underline{\cup}\, \underline{translate}\ (P, p) \,\underline{\cup}\, \underline{translate}\ (P, 2p) \,\underline{\cup} \ldots\ \underline{translate}\ (P, (n-1)p)\{pn\}$$

From the definition of *unfold*

$$N{:}P(p) = unfold\ (N{:}P\{p\}, q).$$

□

3.1.4 Set Theoretic Operators

As with aperiodic events, algebraic set operators are essential in expressing variety of temporal events. Through these operators users can express complex events in terms of simple events. Temporal queries can be formulated with ease using the set

theoretic operators because even the most naive users are familiar with the straightforward semantics of the set theoretic operators.

Remember that the interval algebra is not closed under the set of all intervals [All 83, Ga 88] which causes data duplication over a set of tuples for a single object. This in turn causes data integrity problems. Temporal elements however are closed under the set theoretic operators. The result of any algebra expression is also a temporal element. On one hand this solves the data duplication and the associated problems in the data model, because an object will be stamped by a single temporal element (itself is a set of intervals) at all times. On the other hand the resulting data model is not in the 1NF anymore. From the ease of expressing queries point of view, the non-1NF model provides user with a more intuitive and natural query language than the 1NF models, because user is given a more clear picture of data.

Periodic elements support all set theoretic operators. To our knowledge, none of the existing periodic types offer a set algebra including all set theoretic operators. Furthermore the set of all periodic elements are closed under set theoretic operator as shown by the following three lemmas. The set theoretic operations on periodic elements are illustrated in Figure 3.4.

Lemma 3.5 (Union)

The union $\alpha \cup \beta$ *of two periodic elements* α *and* β *is also a periodic element.*

□

Proof

Let $\alpha = N_1{:}P_1\{p_1\}$ and $\beta = N_2{:}P_2\{p_2\}$. Then

$$\alpha \cup \beta = N_1 : P_1\{p_1\} \cup N_2 : P_2\{p_2\}.$$

Show that $N_1{:}P_1\{p_1\} \cup N_2{:}P_2\{p_2\}$ is a periodic element. Assume without loss of generality that $\underline{begin}\ (P_1) \leq \underline{begin}\ (P_2)$. Let $N_3{:}P_3\{p_1\} = right\ (N_1{:}P_1\{p_1\}, d)$ where

$$N_3 = N_1 \cup \bigcup_{i=1}^{k} \underline{translate}\ (P_1, (i-1)p_1) \cup (\underline{translate}\ (P_1, kp_1) \cap [\underline{begin}\ \underline{translate}\ (P, kp_1)), \underline{translate}\ (\underline{begin}\ (P_1), d)]),$$

$$P_3 = [\underline{translate}\ (\underline{begin}\ (P_1), d), \underline{translate}\ (\underline{begin}\ (P_1), d+p_1)] \cap (\underline{translate}\ (P, kp_1) \cup \underline{translate}\ (P_1, (k+1)p_1))$$

where

$k = \lfloor d / p_1 \rfloor$ and $d =$ ***distance*** $(\underline{begin}\ (P_2), \underline{begin}\ (P_1))$ by Definition 3.4.

Replace $N_1{:}P_1\{p_1\}$ with $N_3{:}P_3\{p_1\}$, since $N_1{:}P_1\{p_1\} = N_3{:}P_3\{p_1\}$ by the right shift rule (Lemma 3.1). Then

$$\alpha \cup \beta = N_3 : P_3\{p_1\} \cup N_2 : P_2\{p_2\}.$$

Let $N_3 : P_4\ \{lcm\ (p_1, p_2)\} = unfold\ (N_3 : P_3\ \{p_1\}, lcm\ (p_1, p_2)/p_1)$ where

$P_4 = \bigcup_{i=1}^{n} \underline{translate}\ (P_3, (i-1)p_1)$ and $n = lcm\ (p_1, p_2)/p_1$ by □

. Since $N_3 : P_3\ \{p_1\} = N_3 : P_4\ \{lcm\ (p_1, p_2)\}$ by □

, replace $N_3 : P_3\ \{p_1\}$ with $N_3 : P_4\ \{lcm\ (p_1, p_2)\}$. Similarly let $N_2 : P_5\ \{lcm\ (p_1, p_2)\} = unfold\ (N_2 : P_2\ \{p_2\}, lcm\ (p_1, p_2)/p_2)$ where $P_5 = \bigcup_{i=1}^{n} \underline{translate}\ (P_2, (i-1)p_2)$ and $n = lcm\ (p_1, p_2)/p_1$ by □

. Since $N_2 : P_2\ \{p_2\} = N_2 : P_5\ \{lcm\ (p_1,p_2)\}$ by □

, replace $N_2 : P_2\ \{p_2\}$ with $N_2 : P_5\ \{lcm\ (p_1, p_2)\}$ (Note that any common multiplier of p_1 and p_2 would suffice for proving the lemma, however $lcm\ (p_1, p_2)$ is chosen to minimize redundancy.):

$$\alpha \cup \beta = N_3 : P_4\{lcm\ (p_1,p_2)\} \cup N_2 : P_5\{lcm\ (p_1,p_2)\}$$

By Definition 3.1,

$$\alpha \cup \beta = N_3 \underline{\cup} P_4 \underline{\cup}\, \underline{translate}\ (P_4,\ lcm\ (p_1,\ p_2)) \underline{\cup}\, \underline{translate}\ (P_4,\ 2lcm\ (p_1,\ p_2)) \underline{\cup} \ldots \underline{\cup} N_2 \underline{\cup} P_5 \underline{\cup}\, \underline{translate}\ (P_5,\ lcm\ (p_1,\ p_2)) \underline{\cup}\, \underline{translate}\ (P_5,\ 2lcm\ (p_1,\ p_2)) \underline{\cup} \ldots$$

After regrouping the terms,

$$\alpha \cup \beta = N_3 \underline{\cup} N_2 \underline{\cup} P_4 \underline{\cup} P_5 \underline{\cup}\, \underline{translate}\ (P_4,\ lcm\ (p_1,p_2)) \underline{\cup}\, \underline{translate}\ (P_5,\ lcm\ (p_1,p_2)) \underline{\cup}\, \underline{translate}\ (P_4,\ 2lcm\ (p_1,p_2)) \underline{\cup}\, \underline{translate}\ (P_5,\ 2lcm\ (p_1,p_2)) \underline{\cup} \ldots$$

Hence by Definition 3.1,

$$\alpha \cup \beta = N_3 \underline{\cup} N_2 : P_4 \underline{\cup} P_5\ \{lcm\ (p_1,\ p_2)\}.$$

□

Lemma 3.6 (Intersection)

The intersection $\alpha \cap \beta$ of two periodic elements α and β is also a periodic element.

□

Proof

Let $\alpha = N_1{:}P_1\{p_1\}$ and $\beta = N_2{:}P_2\{p_2\}$. Then

$$\alpha \cap \beta = N_1 : P_1\{p_1\} \cap N_2 : P_2\{p_2\}.$$

Show that $N_1{:}P_1\{p_1\} \cap N_2{:}P_2\{p_2\}$ is a periodic element. Assume without loss of generality that $\underline{begin}\ (P_1) \leq \underline{begin}\ (P_2)$. Let $N_3{:}P_3\{p_1\} = right\ (N_1{:}P_1\{p_1\},\ d)$ where $N_3 = N_1 \underline{\cup} \bigcup_{i=1}^{k} \underline{translate}\ (P_1,\ (i-1)p_1) \underline{\cup} (\underline{translate}\ (P_1,\ kp_1) \underline{\cap} [\underline{begin}\ \underline{translate}\ (P,\ kp_1)),$

$\underline{translate}\ (\underline{begin}\ (P_1), d)]), P_3 = [\underline{translate}\ (\underline{begin}\ (P_1), d), \underline{translate}\ (\underline{begin}\ (P_1), d+p_1)] \underline{\cap}\ (\underline{translate}\ (P, kp_1)\ \underline{\cup}\ \underline{translate}\ (P_1, (k+1)p_1))$ where $k = \lfloor d / p_1 \rfloor$ and $d =$ ***distance*** $(\underline{begin}\ (P_2), \underline{begin}\ (P_1))$ by Definition 3.4.

Replace $N_1{:}P_1\{p_1\}$ with $N_3{:}P_3\{p_1\}$, since $N_1{:}P_1\{p_1\} = N_3{:}P_3\{p_1\}$ by the right shift rule (Lemma 3.1):

$$\alpha \cap \beta = N_3 : P_3\{p_1\} \cap N_2 : P_2\{p_2\}.$$

Let $N_3 : P_4\ \{lcm\ (p_1, p_2)\} = unfold\ (N_3 : P_3\ \{p_1\}, lcm\ (p_1, p_2)/p_1)$ where $P_4 = \underline{\bigcup}_{i=1}^{n} \underline{translate}\ (P_3, (i-1)p_1)$ and $n = lcm\ (p_1, p_2)/p_1$ by Definition 3.7. Since $N_3 : P_3\ \{p_1\} = N_3 : P_4\ \{lcm\ (p_1, p_2)\}$ by Lemma 3.4, replace $N_3 : P_3\ \{p_1\}$ with $N_3 : P_4\ \{lcm\ (p_1, p_2)\}$. Similarly let $N_2 : P_5\ \{lcm\ (p_1, p_2)\} = unfold\ (N_2 : P_2\ \{p_2\}, lcm\ (p_1, p_2)/p_2)$ where $P_5 = \underline{\bigcup}_{i=1}^{n} \underline{translate}\ (P_2, (i-1)p_2)$ and $n = lcm\ (p_1, p_2)/p_1$ by Definition 3.7. Since $N_2 : P_2\ \{p_2\} = N_2 : P_5\ \{lcm\ (p_1,p_2)\}$ by Lemma 3.4, replace $N_2 : P_2\ \{p_2\}$ with $N_2 : P_5\ \{lcm\ (p_1, p_2)\}$ (Note that any common multiplier of p_1 and p_2 would suffice for proving the lemma, however $lcm\ (p_1, p_2)$ is chosen to minimize redundancy.):

$$\alpha \cap \beta = N_3 : P_4\{lcm\ (p_1,p_2)\} \cap N_2 : P_5\{lcm\ (p_1,p_2)\}$$

By Definition 3.1,

$$\alpha \cap \beta = (N_3\ \underline{\cup}\ P_4\ \underline{\cup}\ \underline{translate}\ (P_4, lcm\ (p_1,p_2))\ \underline{\cup}\ \underline{translate}\ (P_4, 2lcm\ (p_1,p_2))\ \underline{\cup} \ldots)\ \underline{\cap}\ (N_2\ \underline{\cup}\ P_5\ \underline{\cup}\ \underline{translate}\ (P_5, lcm\ (p_1,p_2))\ \underline{\cup}\ \underline{translate}\ (P_5, 2lcm\ (p_1,p_2))\ \underline{\cup} \ldots)$$

Regroup and simplify the terms:

$$\alpha \cap \beta = (N_3\ \underline{\cap}\ N_2)\ \underline{\cup}\ (P_4\ \underline{\cap}\ P_5)\ \underline{\cup}\ (\underline{translate}\ (P_4, lcm\ (p_1,p_2))\ \underline{\cap}\ \underline{translate}\ (P_5, lcm\ (p_1,p_2)))\ \underline{\cup}\ (\underline{translate}\ (P_4, 2lcm\ (p_1,p_2))\ \underline{\cap}\ \underline{translate}\ (P_5, 2lcm\ (p_1,p_2)))\ \underline{\cup} \ldots$$

Hence by Definition 3.1,

$$\alpha \cap \beta = N_3 \underline{\cup} N_2 : P_4 \underline{\cup} P_5 \{lcm\ (p_1, p_2)\}.$$

□

Lemma 3.7 *(Complementation)*

The complement $\sim \alpha$ *of a periodic element* α *is a periodic element.*

□

Proof

Let $\alpha = N{:}P\{p\}$. Assume that N and P are in canonical forms. First write the periodic element $N{:}P\{p\}$ in terms of its constituent intervals as follows:

$$N{:}P\{p\} = N \underline{\cup} P \underline{\cup} \underline{translate}\ (P, p) \underline{\cup} \underline{translate}\ (P, 2p) \underline{\cup} \ldots$$

Let $N = [b_1, e_1] \underline{\underline{\cup}} [b_2, e_2] \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} [b_n, e_n]$ and $P = [s_1, f_1] \underline{\underline{\cup}} [s_2, f_2] \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} [s_m, f_m]$. Then

$$N{:}P\{p\} = [b_1, e_1] \underline{\underline{\cup}} [b_2, e_2] \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} [b_n, e_n] \underline{\underline{\cup}} [s_1, f_1] \underline{\underline{\cup}} [s_2, f_2] \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} [s_m, f_m] \underline{\underline{\cup}} [\underline{translate}\ (s_1, p), \underline{translate}\ (f_1, p)] \underline{\underline{\cup}} [\underline{translate}\ (s_2, p), \underline{translate}\ (f_2, p)] \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} [\underline{translate}\ (s_m, p), \underline{translate}\ (f_m, p)] \underline{\underline{\cup}} \ldots$$

Note that the intervals in the above expression are mutually exclusive, i.e., non-overlapping, the complement of $N{:}P\{p\}$ can be written as the set of instants not included in $N{:}P\{p\}$. Since $6 = [0, \infty)$ and $\sim N{:}P\{p\} = 6 \underline{-} N{:}P\{p\}$, we write

$$\sim N{:}P\{p\} = [0, b_1] \underline{\underline{\cup}} [e_1, b_2] \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} [b_n, s_1] \underline{\underline{\cup}} [f_1, s_2] \underline{\underline{\cup}} [f_2, s_2] \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} [f_m, \underline{translate}\ (s_1, p)] \underline{\underline{\cup}} [\underline{translate}\ (f_1, p), \underline{translate}\ (s_2, p)] \underline{\underline{\cup}} [\underline{translate}\ (f_2, p), \underline{translate}\ (s_3, p)] \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} [\underline{translate}\ (f_m, p), \underline{translate}\ (s_1, 2p)] \underline{\underline{\cup}} \ldots$$

$$\sim N{:}P\{p\} = ([0, b_1] \,\underline{\underline{\cup}}\, [e_1, b_2] \,\underline{\underline{\cup}}\, \dots \,\underline{\underline{\cup}}\, [b_n, s_1]) \,\underline{\underline{\cup}}\, ([f_1, s_2]) \,\underline{\underline{\cup}}\, [f_2, s_2] \,\underline{\underline{\cup}}\, \dots \,\underline{\underline{\cup}}\, [f_m, \underline{\underline{translate}}\ (s_1, p)]) \,\underline{\underline{\cup}}\, ([\underline{\underline{translate}}\ (f_1, p), \underline{\underline{translate}}\ (s_2, p)] \,\underline{\underline{\cup}}\, [\underline{\underline{translate}}\ (f_2, p), \underline{\underline{translate}}\ (s_3, p)] \,\underline{\underline{\cup}}\, \dots \,\underline{\underline{\cup}}\, [\underline{\underline{translate}}\ (f_m, p), \underline{\underline{translate}}\ (s_1, 2p)]) \,\underline{\underline{\cup}}\, \dots$$

From the definition of periodic elements

$$\sim N{:}P\{p\} = ([0, b_1] \,\underline{\underline{\cup}}\, [e_1, b_2] \,\underline{\underline{\cup}}\, \dots \,\underline{\underline{\cup}}\, [b_n, s_1]) : ([f_1, s_2]) \,\underline{\underline{\cup}}\, [f_2, s_2] \,\underline{\underline{\cup}}\, \dots \,\underline{\underline{\cup}}\, [f_m, \underline{\underline{translate}}\ (s_1, p)]) \{p\}.$$

□

Theorem 3.1

The set of all periodic elements 2 is closed under set theoretic operators union (∪), intersection (∩), and complement (~).

□

Proof

The proof follows from Lemma 3.5, Lemma 3.6, and Lemma 3.7.

□

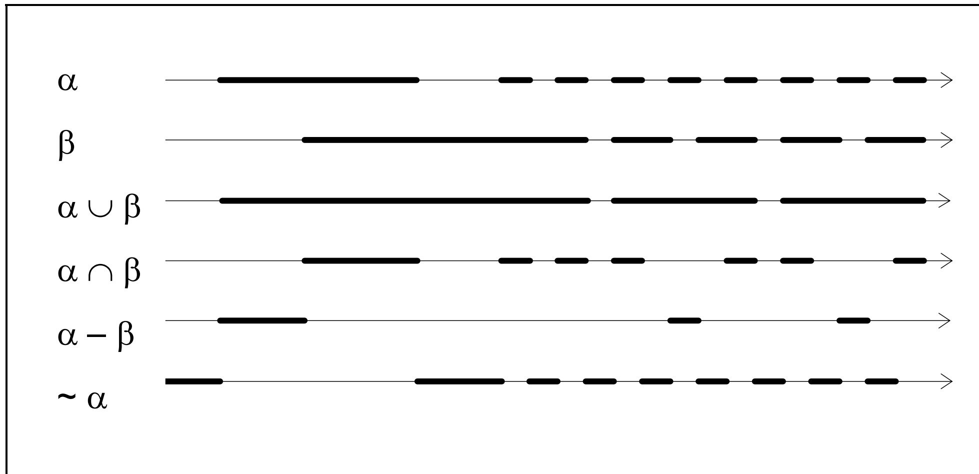

Figure 3.4: The set theoretic operators on periodic elements.

The closure property also holds true between aperiodic and periodic, and aperiodic and aperiodic events, since aperiodic events are defined as a special case of periodic events.

Example 3.2

Let $\alpha = [0, 8] : [24, 26]\ \{12\}$ and $\beta = [5, 12] : [24, 27]\ \{8\}$

$$\alpha \cup \beta = [0, 12] : [24, 27, [32, 35], [36, 38], [40, 43]\ \{24\}.$$

$$\sim (\alpha \cup \beta) = (12, 24), (27, 32): (35, 36), (38, 40), (43, 48)\{24\}.$$

□

Let *pp* and *app* stand for *periodic_part* and *aperiodic_part* functions. Then the union and intersection operators are defined as follows:

Definition 3.8 (Union Algorithm)

Let $\alpha = M : P\ \{p\}$ *and* $\beta = N : Q\ \{q\}$ *be 2 periodic elements. Assuming* $\underline{begin}\ (P)$ ***before*** $\underline{begin}\ (Q) = true$*, the union* $\alpha \cup \beta$ *is computed by the following formula:*

$$\begin{aligned} \alpha \cup \beta = {} & app\ (right\ (\alpha, \mathbf{distance}\ (\underline{begin}\ (P), \underline{begin}\ (Q)))) \underline{\cup} N : \\ & pp\ (unfold\ (right\ (\alpha, \mathbf{distance}\ (\underline{begin}\ (P), \underline{begin}\ (Q))), lcm(p, q)/p)) \underline{\cup} \\ & pp\ (unfold\ (\beta, lcm(p, q)/q)))\ \{lcm\ (p, q)\}. \end{aligned}$$

□

The correctness of this definition follows from Lemma 3.5. When α and β are temporal elements, the union ∪ on periodic elements reduces to the union $\underline{\cup}$ on elements. Similarly if α and β are intervals, then the union $\underline{\cup}$ on elements reduces the union $\underline{\underline{\cup}}$ of intervals. The intersection of periodic elements is defined similarly to union:

Definition 3.9 (Intersection Algorithm)

Let $\alpha = M{:}P\{p\}$ *and* $\beta = N{:}Q\{q\}$ *be 2 periodic elements. Assuming* $\underline{begin}\ (P)$ ***before*** $\underline{begin}\ (Q) = true$*, the intersection* $\alpha \cap \beta$ *is computed by the following formula:*

$$\begin{aligned} \alpha \cap \beta = {} & app\ (right\ (\alpha, \mathbf{distance}\ (\underline{begin}\ (P), \underline{begin}\ (Q)))) \underline{\cap} N : \\ & pp\ (unfold\ (right\ (\alpha, \mathbf{distance}\ (\underline{begin}\ (P), \underline{begin}\ (Q))), lcm(p, q)/p)) \underline{\cap} \\ & pp\ (unfold\ (\beta, lcm(p, q)/q)))\ \{lcm\ (p, q)\}. \end{aligned}$$

□

The correctness of this definition follows from Lemma 3.6. The definition of complementation was given in Lemma 3.7. Thus all of the set theoretic operators on periodic elements can be implemented using the formatting functions and other operators. See Appendix for implementations of these operators in O_2C.

3.1.5 Relational Operators

Temporal relationships between intervals are studied in detail [All 83] and are adopted by temporal models for aperiodic data. We have extended these operators to temporal elements in Chapter 2. These operators can be further extended to periodic elements to capture similar temporal relationships between periodic events. The following are relationships defined for temporal events α and β:

- α *before* β = *true* *if end* (α) ***before*** *begin* (β) = *true.*
- α *start* β = *true* *if begin* (α) = *begin* (β).
- α *finish* β = *true* *if end* (α) = *end* (β).
- α *contain* β = *true* *if* $\alpha \setminus \beta \neq \emptyset$.
- α *meet* β = *true* *if end* (α) + *1* = *begin* (β).
- α *overlap* β = *true* *if* $\alpha \cap \beta \neq \emptyset$.
- α *equal* β = *true* *if* $\alpha \setminus \beta = \beta \setminus \alpha = \emptyset$.

The functions *begin* and *end* return the first and the last instant of a periodic element (for strictly and partially periodic events function *end* returns ∞). The inverse relationship operators (*after, startedby, finishedby, containedby, metby, overlappedby*) are defined likewise. These operators reduce to their aperiodic counterparts for aperiodic operands. Let α and β be periodic elements, N and M temporal elements, then

$$(\alpha = N, \beta = M) \Rightarrow \alpha \textit{ operand } \beta = N \textit{ \underline{operand} } M$$

for *operand* $\in$ *{before, after, ..., equal}* where <u>*operand*</u> stands for an operand of temporal element type. The correctness follows from the definitions of temporal relationships operators on periodic elements (*operand*) and temporal elements (<u>*operand*</u>).

3.2 Relative Periodic Time

Relative events such as the templates in *Example 1.1* are expressed similarly to absolute events. The relative temporal events have been neglected to a great extent in the database and artificial intelligence research. Most effort has been devoted to developing absolute periodic time models. This lack of research into relative time might be stemming from the assumption that relative time can be represented and reasoned about similar to absolute time. We believe that this assumption is not totally correct. Even though absolute and relative time are similar, their semantics are not the same. For example, an absolute event is expressed with references to absolute time instants on a time line, while relative events are expressed using distance information with respect to the unknown beginning of the relative event. While temporal relationship (comparison) operators, set theoretic operators etc. are defined for absolute time, these operators do not readily apply to relative events. We believe the temporal transformations, temporal relationships, set theoretic operators, and the specialized operators should be redefined clearly to capture the semantics of relative time properly. It is also necessary to define conversion between relative and absolute time since such conversions also occur in real world applications.

The main distinction between an absolute and a relative event is that the beginning of a relative event is unknown. An absolute event is described in references to the actual time points on the time line. On the other hand a relative event is described with respect to an undetermined beginning point. We will call the unknown beginning of a relative event *virtual reference point* (*vrp*) and denote it by the bullet symbol •. Thus a relative event is expressed with respect to its vrp. As with absolute time in the previous section, we will introduce aperiodic relative temporal types, *relative interval, relative element* first. The relative periodic time represented by the *relative periodic element* type is then introduced.

3.2.1 Relative Intervals

A relative interval *[•, d]* is an interval with an unknown beginning time *•*. $d \in 0$ is the length of the relative interval specified with respect to the beginning *•*. Note that *d* is not an instant as in absolute time, but a duration (representing *d* time units). For example, *[•, 5]* represent a duration of length 5. As before, the behavior of the types is given as four set of operators or functions. Note that some operators on absolute intervals do not apply to relative intervals anymore.

Specialized operators:. Let *[•, d]* be a relative interval

- *begin ([•, d]) = •,*
- *end ([•, d]) = •,*
- *length ([•, d]) = d,*
- *distance* $([\bullet, d_1], [\bullet, d_2])$ *is undefined.*

Since the beginning of a relative interval is unknown the end of a relative interval is also unknown. By the same token the distance between two relative intervals are unknown.

Temporal transformations: A relative interval can be scaled as follows. Let s be an integer. Then

- *scale ([•, d], s) = [•, ds]*
- *translate ([•, d], e) = (•, •),[e, e+d]*

In general a relative interval can not be translated because the beginning of the interval is not known. However the translations with respect to the *vrp* make sense in the presence of more than one relative intervals whose *vrp*s are the same. That is to say that in some application domains such as multimedia where at the design time the beginning of a multimedia presentation is unknown, a relative event can be translated with respect to the other events. For example, if sound stream is to start 10 second after the start of the video stream in the presentation, the 10 second delay is a temporal translation with respect to the beginning of the presentation. In the

definition above, the first interval ***(•, •)*** in ***(•, •)****,[e, e+d]* serves as a reference point and is not part of the event itself (which is also obvious from the representation itself since both sides of the interval ***(•, •)*** is open and the beginning and end time of the interval is the same).

Relational Operators: We skip the definition of the temporal relationships between relative events, since it is a special case of the temporal relationships between relative temporal elements which is discussed in Section 3.2.2.

Set theoretic operators: The set theoretic operators can not be applied for any for any two unrelated temporal events. However it is possible to introduce set theoretic operators between two relative events expressed with respect to the same *vrp*.

- $[\bullet, d_1] \,\underline{\underline{\cup}}\, [\bullet, d_2] = [\bullet, max(d_1, d_2)]$
- $[\bullet, d_1] \,\underline{\underline{\cap}}\, [\bullet, d_2] = [\bullet, min(d_1, d_2)]$
- $\underline{\underline{\sim}}\, [\bullet, d_1] = ((\bullet, \bullet), (d_1, \infty))$

3.2.2 Relative Temporal Elements

Relative temporal elements are defined similarly to absolute temporal elements. A relative element N is given as

$$N = ([\bullet, e_1], [s_2, e_2], ..., [s_k, e_k]) = [\bullet, e_1] \,\underline{\underline{\cup}}\, [s_2, e_2] \,\underline{\underline{\cup}} ... \underline{\underline{\cup}}\, [s_k, e_k].$$

for $k \geq 1$.

The element *([•, 3], [7, 9]),* for instance, represents a relative aperiodic event that first happens for 3 units of time, then happens for 2 units of time after 4 units of time. By definition every relative interval is a relative element denoted by the interval itself. *Canonical* form for relative elements is defined similarly. For example, the relative element $[\bullet, 8] \,\underline{\underline{\cup}}\, [7, 9]$ is not in canonical form whereas the relative element [•, 8] is. Since *canonical* function is already defined for absolute temporal element, we don't repeat the definition here again. As before we introduce the behavior of

relative periodic elements through four sets of operators: specialized operators, temporal transformations, temporal relationship operators, and set theoretic operators.

Specialized Operators: We give the definitions of the specialized operators for relative time.

- first_interval $([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]) = [\bullet, e_1]$.
- select_interval $(([\bullet, e_1], [b_2, e_2], ..., [b_k, e_k], ..., [b_n, e_n]), k) = [\bullet, e_k - b_k]$.
- last_interval $([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]) = [\bullet, e_n - b_n]$.
- length $([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]) = e_1 + e_2 - b_1 + ... + e_n - b_n$.
- number_of_intervals $([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]) = n$.
- begin $([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]) = \bullet$.
- end $([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]) = e_n$.
- distance *(N, M) is undefined.*

Relational operators: The temporal relationships between relative periodic element are defined the same as those between absolute temporal element in Table 2.3. Remember that the begin, end, **before**, **after**, etc. are redefined in this section for relative time.

Set theoretic operators: Let *R* and *Q* be two relative elements. The relative set theoretic operators (indicated by ‘ symbol below) can be computed by converting the operands to absolute operands via *anchor* functions, and then applying the absolute set theoretic operator. The absolute result can then be converted to a relative value via *unanchor* function:

- $R \cup' Q$ = unanchor (anchor $(R, 0)$ $\cup$ anchor $(Q, 0)$).
- $R \cap' Q$ = unanchor (anchor $(R, 0)$ $\cap$ anchor $(Q, 0)$).

The complementation ($\sim$) is given in a straightforward way as follows:

- $\sim ([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]) = ((\bullet, \bullet), [e_1, b_2], ..., [e_n, \infty])$

$$\simeq ((\bullet, \bullet), [b_2, e_2], ..., [b_n, e_n]) = ([\bullet, b_2], [e_2, b_3], ..., [e_n, \infty])$$

Temporal transformations:

- <u>*scale*</u> $(([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]), s) = ([\bullet, e_1 s], [b_2 s, e_2 s], ..., [b_n s, e_n s])$
- <u>*translate*</u> $(([\bullet, e_1], [b_2, e_2], ..., [b_n, e_n]), d) = ((\bullet, \bullet), [d, e_1+d], ..., [b_n+d, e_n+d])$

For s > 1, the scale function reduces the size of the interval in relative element by the factor s. For s < 1, the scale function increases the size of the interval by the factor s. Translation is performed with respect to *vrp* of the relative element.

3.2.3 Relative Periodic Elements

A relative periodic event is a periodic event with an unspecified beginning time. The intervals in the relative periodic element is specified with respect to this beginning. For example, an event happening for 2 units of time in every 8 units of time is represented by the following relative periodic element $[\bullet, 2]\ \{8\}$.

Definition 3.10 (Relative periodic element)

A relative periodic element α' is a periodic element α whose beginning is unknown (denoted by $\bullet$):

$$\alpha' = [\bullet, e_1], [b_2, e_2], ..., [b_k, e_k] : [b_{k+1}, e_{k+1}], ..., [b_n, e_n]\{p\}$$

□

3.2.4 Conversion between Relative and Absolute time

As events evolve in time an absolute event may become a relative event or a relative event may become an absolute event. For example, consider a meeting of 2 hour long. Assume that the meeting is schedule at time instant *t*, then this absolute event is denoted by $[t, t+2]$. If this meeting needs to be rescheduled for some reason, the meeting should be stored as a relative event until the new meeting time is determined.

Until that time it can be stored as *[•, 2]* which tells us that the meeting lasts two hours but (the beginning) time is unknown. If the meeting is rescheduled to $t+10$, it is stored as $[t+10, t+12]$ for some t. The functions *anchor* and *unanchor* convert between absolute and relative time.

Definition 3.11 (Unanchor)

Given a periodic element, the function unanchor returns the corresponding relative periodic element:

$$unanchor\ ([t_0, t_1], \ldots, [t_{k-1}, t_k] : [l_0, l_1], \ldots, [l_{n-1}, l_n]\{p\}) =$$
$$[\bullet, t_1 - t_0], \ldots, [t_{k-1} - t_0, t_k - t_0] : [l_0 - t_0, l_1 - t_0], \ldots, [l_{n-1} - t_0, l_n - t_0]\{p\}.$$

□

Definition 3.12 (Anchor)

Given a relative periodic element α*, the anchor function anchor* (α, t) *returns an absolute event corresponding to the happening of* α *at time instant* t*:*

$$anchor([\bullet, a_1],\ldots,[a_{k-1}, a_k]:[p_0, p_1],\ldots,[p_{n-1}, p_n]\{p\}, t) =$$
$$[t, a_1+t], \ldots, [a_{k-1}+t, a_k+t] : [p_0+t, p_1+t], \ldots, [p_{n-1}+t, p_n+t]\{p\}.$$

□

Note that in the strictest sense, $t_1 - t_2$, and $t_1 + d$ correspond to ***distance*** (t_1, t_2) and ***translate*** (t_1, d).

The function extracts (unanchors) the temporal pattern from an absolute periodic event. The function fixes (anchors) the unknown beginning of relative events to a given instant. Relative elements, relative intervals and relative periodic elements are defined similarly to their absolute counterparts.

Example 3.3

- *unanchor ([5, 9]) = [•, 4].*
- *unanchor ([2, 7], [12, 15]) = [•, 5], [10, 13].*
- *unanchor ([2, 6]{7}) = [•, 4]{7}.*
- *unanchor ([3, 5] : [7, 9]{4}) = [•, 2] : [5, 7] {4}.*
- *anchor ([•, 4], 6) = [6, 10].*
- *anchor ([•, 5], [10, 13], 2) = [2, 7], [12, 15].*
- *anchor ([•, 4]{7}, 2) = [2, 6]{7}.*
- *anchor ([•, 2] : [5, 7] {4}, 3) = [3, 5] : [7, 9]{4}.*

□

Specialized operators: Specialized functions on absolute periodic elements apply to relative periodic elements, too. (See 3.1.2 for details)

Example 3.4

Let α = [•, 2] : [5, 9] {12} *be a relative periodic event. Then,*

- *aperiodic_part(α) = [•, 2],*
- *periodic_part(α) = [•, 4],*
- *period(α) = 12.*

□

The result of a specialized operator is always a relative event. Formatting functions can also be applied to relative periodic events in a similar fashion to absolute periodic events. (See 3.1.3 for definitions)

Example 3.5

Let α = [•, 4] : [6, 8], [10, 12]{8} be a relative periodic event. Then

right (α, 4) = [•, 4], [6, 8] : [10, 12], [14, 16]{8}.

left (α, 4) = [•, 2] : [2, 4], [6, 8]{8}.

fold (α, 2)= [•, 4] : [6, 8]{4}.

unfold (α, 2) = [•, 4] : [6, 8], [10, 12], [14, 16], [18, 20]{8}.

□

Set theoretic operators: Set theoretic operations also applies to relative periodic elements the only difference is being that the beginning of the both operands is the same, namely the • symbol. The closure property also holds true for relative periodic elements. Let *R* and *Q* be two relative periodic elements. The relative set theoretic operators (indicated by ‘ symbol below) can be computed by converting the operands to absolute value via *anchor* function, and then applying the absolute set theoretic operators. The absolute result can then be converted to a relative value via *unanchor* function:

- *R ∪‘ Q = unanchor (anchor (R, 0) ∪ anchor (Q, 0)).*
- *R ∩‘ Q = unanchor (anchor (R, 0) ∩ anchor (Q, 0)).*

The complement (≃) is given in a straightforward way as follows:

- $\simeq$ ([•, e_1],[b_2, e_2], ..., [b_k, e_k]: [b_{k+1}, e_{k+1}], ..., [b_n, e_n]{p}) =
  ((•, •), [e_1, b_2], ..., [e_k, b_{k+1}]: [e_{k+1}, b_{k+2}], ..., [e_n, b_{k+1}+p]).
- $\simeq$ ((•, •),[b_2, e_2], ..., [b_k, e_k]: [b_{k+1}, e_{k+1}], ..., [b_n, e_n]{p}) =
  ([•, b_2), [e_2, b_3], ..., [e_k, b_{k+1}]: [e_{k+1}, b_{k+2}], ..., [e_n, b_{k+1}+p]).

Example 3.6

Let α = [•, 1]{2} and β = [•, 2]{3} be two relative periodic event. Then

- $\alpha \cup \beta = [\bullet, 5]\{6\}$
- $\alpha \cap \beta = [\bullet, 1], [4, 5]\{6\}$
- $\sim \alpha = (\bullet, \bullet): [1, 2]\{2\}$

□

3.3 Finite Periodic Time

The aperiodic part of a partially-periodic event may be a periodic event happened for a finite but large number of time. For example the schedule of a part time employee who working seven times a week for 5 years can contain 5year x 52weeks x 5days=1300 intervals. Obviously the explicit representation of these intervals in the representation is redundant. We offer the following representation to deal with such problems:

Definition 3.13

A finite periodic event $P\{p\}_q$ is a portion of a strictly periodic event $P\{p\}$ happening until a given time instant $q \geq min(P)$.

$$P\{p\}_q = P\{p\} \cap [begin\ (P), q].$$

□

The finite periodic event $P\{p\}_q$ is computed by the following formula:

$$P\{p\}_q = (\bigcup_{i=1}^{k} \underline{translate}\ (P, (i-1)p)) \underline{\cup} (\underline{translate}\ (P, kp) \underline{\cap} [\underline{begin}\ (P), q])$$

where $k = \lfloor q/p \rfloor$

There are two advantages for using finite periodic elements to represent finite periodic events. First, it is intuitive, and simpler: $P\{p\}_q$ means that event P happens at cycles of p until q. Second, it is time/space efficient than an explicit presentation. The formula $P\{p\}_q$ requires $2n+2$ memory (assuming n intervals in P, plus 2 words for p

and *p*). However the explicit representation $(\bigcup_{i=1}^{k} \underline{translate}(P, (i-1)p)) \underline{\cup} (\underline{translate}(P, kp) \underline{\cap} [\underline{begin}(P), q])$ requires approximately k times more memory.

Having defined finite periodic elements, we now give the definition of a generalized periodic element that can represent a partially-periodic event having finite periodic segments in its aperiodic part.

Definition 3.14 (Generalized periodic element)

A sequence of finite periodic elements N, a temporal element P, and a duration p, is called a periodic element (denoted by N:P{p})

$$N{:}P\{p\} = N \underline{\cup} P\{p\}$$

where

- $N = \bigcup_{i=1}^{n} Q_i\{s_i\}_{qi}$
- $P\{p\} = \bigcup_{i=1}^{\infty} \underline{translate}(P, (i-1)p)$
- *Q_is and P do not contain ∞,*
- $p \geq 0,\ s_i \geq 0,$
- $Q_i\{s_i\}_{qi}$ $\underline{before}$ $P = true,$
- $p \geq$ ***distance*** *(end (P), begin (P)).*

□

The main difference between *Definition 3.14* and Definition 3.1 is the aperiodic part N of periodic event N:P{p} consists of finite periodic elements instead of simple intervals.

Example 3.7

Let α = [1, 2]{4} be a periodic event at time instant 20 extending into future. If the event started to happen in every 8 time units at time instant 300, it whole event is expressed as follows:

$$\alpha = [1, 2]\{4\}_{300}{:}[300, 301]\{8\}.$$

If the event started to happen for a duration 3 time units instead of 2 at time instant 500, then the whole event is expressed as follows:

$$\alpha = [1, 2]\{4\}_{300}, [300, 301]\{8\}_{500}{:}[500, 503]\{8\}.$$

If the event has returned to its original happening pattern [1, 2]{4} at time instant 700, the whole history of the event is given as

$$\alpha = [1, 2]\{4\}_{300}, [300, 301]\{8\}_{500}, [500, 503]\{8\}_{700}{:}[700, 701]\{4\} .$$

□

As with any other periodic element, a periodic element containing any numbers of finite periodic elements can be either absolute or relative. Such a relative periodic element starts with the • symbol as before. Finally, we give the definitions of temporal events.

Definition 3.15

A temporal event represented by a periodic element N:P{p} is called a partially-periodic event (periodic event for short). A partially-periodic event N:P{p} is called

- *an aperiodic event if the periodic part is empty (P = ∅),*
- *a strictly-periodic event if the aperiodic part is empty (N = ∅).*

□

5, [3, 7], ([2, 5], [8, 10]) are examples of aperiodic events, while *[2, 5]{6}* and *([0, 4], [6, 7)):[9,10]{5}* are examples for strictly-periodic and partially-periodic events respectively.

Definition 3.16

A temporal event represented by a relative periodic element N:P{p} is called a relative partially-periodic event (relative periodic event for short). A relative partially-periodic event N:P{p} is called

- *an relative aperiodic event if the periodic part is empty (P = ∅),*
- *a relative strictly-periodic event if the aperiodic part is empty (N = ∅).*

□

3.4 Summary

In this chapter a new temporal type called periodic element for representing and reasoning about periodic events is proposed. Periodic element can represent aperiodic events, strictly-periodic events and partially-periodic events. We have gracefully extended all existing temporal operators of aperiodic time into periodic elements to allow user express queries involving temporal transformations (*translate, scale*), temporal relationships (*before, after, contain, overlap*, etc.), set theoretic operations (*union, intersection, complement, difference*), and specialized operators (*begin, end, length, distance,* etc.). We have proved that the algebra defined on periodic elements is closed under not only temporal transformations but also under set theoretic operations. We have shown how to compute formatting functions and set theoretic operations efficiently. Furthermore, we introduced relative intervals, relative temporal elements and relative periodic element types in Section 3.2. Temporal types for modeling relative time in general. A neglected issue in temporal database research. Their behavior under temporal transformations, set theoretic operations, and the temporal relationships between relative events are studied. We introduced finite

periodic element for representing events happening periodically within a finite duration of time. Lastly, we modified the definition periodic elements to represent events whose aperiodic part is a sequence of finite periodic events.

Chapter 4

Periodic Temporal Data Model

Temporal databases have been one of the focal points of interest in database research in the last decade. However, the issue of periodic events has started to draw attention only recently with the advent of calendars in temporal databases. We believe it is appropriate to say that the field of periodic temporal databases is in its infancy while the research in aperiodic temporal databases especially into relational model has been saturated with dozens of models and languages.

This chapter deals with modeling periodic temporal databases in object-oriented paradigm. There are two basic approaches to incorporating time into a data model. First approach modifies the data model and the language accordingly to deal with time. This approach is the most commonly used approach so far. The tuples or attribute values usually have associated time stamps representing the valid time of the tuple or value. The time-stamps are implicitly handled by the database system. The query language is augmented with new constructs like *while* or *when* by which temporal conditions are specified. The second approach models temporal values as a function of time. An object or an attribute always have a unique value at a given time point. The value itself may be a complex value such as a list or set, but this doesn't change the fact that the value is unique at a given point in time. Thus a temporal attribute can be thought of a set of mappings from time domain into the attribute domain. For aperiodic events the number of mappings in the function is finite assuming the number of intervals in a time stamp is finite. For periodic events the number of mappings is possibly infinite in the light of the fact that periodic events may extend into indefinite future.

The previous work in extending databases with a time dimension is limited in some respects [WD 92]. First, most of the models extend relational model either by tuple versioning or by attribute versioning. It is known that the relational model is

inadequate for capturing the semantics of complex objects arising in many applications. Second, the temporal and non-temporal data are not treated uniformly. New, special-purpose temporal operators such as when [CT 85, Sn 87] are introduced to query temporal dimension of the data. Such non-uniform languages make query formulation and query optimization difficult. Third, the models extended for time usually deal only with a single notion of time, usually a linearly ordered time points or intervals (corresponding to calendar dates). However different applications require different notions of time. For example business applications require a calendars system (discrete), while scientific applications usually deal with high precision time (continuous).

We describe a periodic temporal model and query language to overcome these difficulties. We start with the data model of a commercially available object oriented database management system O_2. O_2 data model supports all basic ingredients of object oriented data model [Cat 94] including object identity, complex object types, encapsulation, inheritance, dynamic binding, etc. A detailed description of O_2 can be found in [Ba 92]. In our approach we use objects to model real world entities and methods to model properties, relationships and operations of objects.

To model temporal aspect of the data we depend on the type system of O_2 data model. We treat time as an abstract data type and define a hierarchy of temporal types to support different notions of time. O_2 supports sets, bags, tuples and lists. These constructs are essential in modeling various temporal types. The temporal behavior of objects are defined by methods relating time to values. The behavior of temporal types are defined by temporal functions corresponding to the operations defined in Chapter 2 and 3. Thus the temporal extension is achieved without modifying the O_2 data model. Since the temporal behavior of objects are uniformly modeled by methods and functions, no special constructs are needed in O_2 query language. Thus we demonstrate that our language meets the limitations that cannot be satisfied by other existing languages. Our approach is general and can be used with any object oriented data model and query language.

This chapter is organized as follows. In Section 4.1 we present the temporal extension to an object oriented model by modeling temporal attributes as a function of time. In Section 4.2, we describe the temporal type hierarchy for representing aperiodic and periodic time. The valid time of entities in the model is discussed in Section 4.3. Temporal constraints enforced by the model are studied in Section 4.4. Finally the chapter is concluded with a summary in Section 4.5.

4.1 Modeling Temporal Data Using Methods

In the non-temporal model, a property of an object is modeled as an attribute that return a snapshot value of the property. Attributes of an object can be time varying. For example a salary attribute can change over time. Such behavior is modeled by methods that maps time points into snapshot values of the attribute.

There are two types of temporal attributes: *Static* and *dynamic*. Static attributes are *time invariant* such as the birth date attribute and never change. Dynamic attributes or *time variant* attributes can change in time. In more advanced applications a dynamic temporal attribute can change periodically.

In our model the type of an attribute could be atomic or complex. Atomic values come from the following values: *integer, real, string, boolean, bits*. Complex values are defined by the *tuple, set, unique set* and *list* constructs.

Temporal data can be modeled at two different levels: object/tuple versioning [NA 87] or attribute versioning [CT 85, EW 90]. As opposed to attribute versioning, object versioning stores the entire versions of the evolving objects. An object-oriented model can support both approaches. We chose the attribute versioning approach because the attributes of the same object may change asynchronously over time. A temporal attribute is given by its temporal assignment defined below.

Definition 4.1

A temporal attribute is a function f: 6 → # where 6 is the time domain i.e. the set of all time instants and # is an atomic or complex attribute domain.

□

With the above definition, the temporal attribute *A* can be given as a set of mappings of the form *(t, a)* where $t \in$ 6 and $a \in$ #. For attributes with unbounded or infinite time domain, there are infinite number of *(t, a)* mappings. It is possible to represent the *f* function with finite number of mappings if the time is specified by a periodic element instead of an instant. We call the finite representation of function *f* the *temporal assignment of attribute A* similar to [G 88].

Definition 4.2 (Temporal assignment)

The temporal assignment of temporal attribute A is a function f: 2 !→ # represented by a finite number of mappings of the form (N:P{p} → a) meaning the value of f during the time N:P{p}:

$$f = \{(\alpha_i, a_i) \mid \alpha_i \in 2, a_i \in \#, \text{for any } i \neq j\ \alpha_i \cap \alpha_j = \emptyset, \text{for any } i \neq j\ a_i \neq a_j\}.$$

□

Then the value at time instant $t \in T$ is given as

$$f(t) = a, \text{ where } \exists (\alpha, a) \text{ s.t. } (\alpha, a) \in f,\ \alpha \text{ contains } t = true.$$

Example 4.1

The salary attribute can be expressed as a temporal assignment:

{ ([1/1/90, 12/31/95]{1year} → $1000),

([1/1/96, 6/31/96] {1year} → $1100),

([7/1/96, 12/31/96] {1year} → $1050) }

The first mapping *([1/1/90, 12/31/95] → $1000)* states that the salary was $1000 between *1/1/90* and *12/31/95.* The second mapping *([1/1/96, 6/31/96] → $1100)* means that the salary is $1100 for the first half of every year starting with year 1996. Similarly, the third mapping means that the salary is $1050 for the second half of every year starting with year 1996.

4.2 Time as a Set of Abstract Data Types

Most temporal data models deal with a single notion of time usually linear discrete time points. This simplistic view might not satisfy the requirements of many different applications. Our approach takes a layered hierarchical approach to modeling time. Temporal types are modeled as abstract data types. (Here we mention the absolute temporal types only. The relative types defined in the previous chapter are treated exactly the same.) At the bottom of this hierarchy is the temporal type *Instant* representing the underlying notion of time which may be continuos or discrete, user defined or built-in (date). The time points represented by *Instant* type. The behavior of *Instant* is given by a set of functions which can be overridden by the user to define different notions of time. The more complex types in the hierarchy is defined in terms of simpler ones as follows:

```
type Interval tuple (    begin: Instant,
                         end: Instant);
type Element list (Interval);
type P_element tuple (   aperiodicpart: Element,
                         periodicpart: Element,
                         period: Duration)
```

Interval is tuple of two attributes *begin* and end. A temporal element is a list of *Intervals* ordered by their beginning time. A periodic element (*P_element*) is a tuple of three attributes. *aperiodic* and *periodic* parts which are of element type and the

period. The type of period is Duration which can be modified by user. For concrete time Duration is defined as integer. The behavior of these temporal types is defined by four sets of functions as discussed in Chapter 2 and 3.

Specialized functions: These function computes some special properties of a temporal event represented by the above temporal types. Here are some of these functions

- *begin*: return the beginning of an event,
- *end*: return the end of an event
- *distance*: computes the distance between two events in terms of number of *Instant*s,
- *length*: computes the duration of an event
- *select_interval*: select the *k*th interval from an event.

Temporal comparisons: These functions compute the temporal relationships between two temporal events as defined by Allen in [All 83] for interval. We further extended these relationships to temporal elements and periodic elements in Chapter 2 and 3.

Set theoretic operations: The operations union, intersection, and complementation and difference was already defined for interval [All 83] and element [G 88]. However we extend the set theoretic operation to periodic elements. Thus it is possible to compute the union of two partially-periodic events efficiently.

Temporal transformations: Temporal transformations of *scale* and *translate* are defined for the above temporal types with the exception that an instant cannot be scaled. It is shown in Chapter 2 and 3 that the above types are closed under temporal transformation. The types of elements and periodic elements are also closed under set theoretic operations.

4.3 Valid time

In a temporal database the past values of an attribute are never discarded when the attribute is changed. The previous values can be queried by users. We introduce a function called *vtime* for computing the valid time of an entity which may be an attribute, object, or a collection of objects. In other words the *vtime* function computes the *lifespan* of a temporal entity.

Definition 4.3

Let $f = \bigcup_{i=1}^{n} (\alpha_i, a_i)$ *be the temporal assignment of a temporal attribute A where* $\alpha_i \in 2,$ $a_i \in$ #*. Then the valid time of attribute A is the time during which A assumes a value in* #*:*

$$vtime\ (A) = \bigcup_{i=1}^{n} \alpha_i\ .$$

□

Example 4.2

Consider the salary attribute in Example 4.1. The valid time for the salary attribute is

$$\begin{aligned} vtime(salary) &= [1/1/90,\ 12/31/95] \cup [1/1/96,\ 6/31/96] \cup [7/1/96,\ 12/31/96] \\ &= [1/1/90,\ \infty). \end{aligned}$$

□

Definition 4.4

Let object o = $\{f_i \mid f_i$ *is temporal assignment of* A_i $1 \le i \le n\}$ *be a set of temporal attributes* $A_1, A_2, \ldots A_n$*. Then the valid time of object o is time during which any of attributes assume a value:*

$$vtime\ (o) = \bigcup_{i=1}^{n} vtime\ (A_i)$$

where vtime (o_i) *is defined in Definition 4.3*

□

Example 4.3

Let john be an employee object with two attributes: empname and salary. Let the temporal assignment for empname be $\{([1/1/90, \infty) \rightarrow \text{"john"})\}$*. The salary attribute is given in Example 4.1. Then the valid time of object john is computed as follows:*

$$\begin{aligned} vtime\ (john) &= vtime\ (empname) \cup vtime\ (salary) \\ &= [1/1/90, \infty) \cup [1/1/90, \infty) \\ &= [1/1/90, \infty). \end{aligned}$$

□

Definition 4.5

Let $S = \{o_i \mid o_i \in C, 1 \le i \le n\}$ *be a set of objects in class C. Then the valid time of S is the time during which any of the attributes of any object in the set assumes a value:*

$$vtime\ (S) = \bigcup_{i=1}^{n} vtime\ (o_i)$$

where vtime (o_i) is defined in Definition 4.4.

□

The valid time of a bag of objects is defined similarly. A *bag* may contain multiple copies of an object. A list imposes an order on a bag of objects. An object may occur more than once in a list. A list is delimited by enclosing the elements of the list between < > symbols.

Definition 4.6

Let $S = <o_1, o_2, ..., o_n> 1 \leq i \leq n>$ *be a list of objects in class C. Then the valid time of S is the time during which any of the attributes of any object in the list assumes a value:*

$$vtime\ (S) = \bigcup_{i=1}^{n} vtime\ (o_i).$$

where vtime (o_i) is defined in Definition 4.4.

□

Since a class is a set of object of the same type specified by the class/type definition, the valid time of a class is computed in the same way it is computed for any set of objects (see *Definition 4.5*), i.e., the valid time of a class is the union of the valid times of objects in the class.

Class hierarchy defines the super/sub class relationships among a set of classes who share data and methods [O 95]. A sub class inherits the attributes and methods of its super-classes and defines new ones uniquely applicable to itself. Objects in a subclass also belong to the super-classes of the class. However the objects of a super-class may or may not belong to a sub class. The valid time of a super-class can be computed by the valid time of its immediate subclasses and the objects belonging to the super-class itself.

Definition 4.7

Let S be a class and $S_1, S_2, \ldots, S_{an}$ be the immediate subclasses of S where each I_s can be the root of another class hierarchy. Let S' be the set of objects belonging uniquely to S and to any sub-classes. Then the valid time S is recursively defined:

$$vtime\ (S) = vtime(S') \cup \bigcup_{i=1}^{n} vtime\ (I_s)$$

□

4.4 Temporal Constraints

To assure the integrity of temporal data stored in the database a set of temporal constraints must be enforced. In relational databases the key must uniquely identify a tuple i.e., no two tuples are allowed to have the same key attribute value. In object oriented databases each object is identified by a unique object id. This idea of unique identification applies to temporal attributes of a temporal databases as follows: A temporal attribute can assume a unique value at any time instant during the lifespan of the attribute. This fact is given in the definition of temporal assignment f of a temporal attribute A:

$$f = \{(\alpha_i, a_i) \mid \alpha_i \in 2, a_i \in \#, \text{for any } i \neq j\ \alpha_i \cap \alpha_j = \varnothing, \text{for any } i \neq j\ a_i \neq a_j\}.$$

The condition $\forall i, j \text{ s.t. } i \neq j\ \alpha_i \cap \alpha_j = \varnothing$ enforces that at any time point there can be at exactly one snapshot value. However this does not prevent the unique value to be a set of complex values of objects. Other temporal constraints follow from the definitions:

- Let o be an object of tuple type $A_1, A_2, \ldots, A_n$ where A_i are attributes. Then *vtime (o) contains vtime* (A_i) *for* $1 \leq i \leq n$.
- Let o be an object of set type $\{A\}$ attribute A. Then *vtime (o) contains vtime (A).*

- Let *o* be an object of list type $<A_1, A_2, .., A_n>$ where A_i are attributes. Then *vtime (o) contains vtime* (A_i) *for* $1 \leq i \leq n$.
- Let S be a class and $S_1, S_2, ..., S_n$ be the immediate subclasses of S. Then *vtime (S) contains vtime* (S_i) *for* $1 \leq i \leq n$.

These for temporal constraints are naturally satisfied by the data model itself by definition of the *vtime* function and need not be enforced directly.

4.5 Summary

In this Chapter, a periodic temporal object-oriented data model for modeling periodic temporal events in databases is presented. The model describes time as an abstract data model. Temporal types are introduced as a type hierarchy. The behavior of temporal types are defined by functions. The temporal data is treated as a method mapping time points to attribute values. The temporal and non-temporal attributes are treated uniformly. We believe that our approach overcomes the limitations of existing temporal models while allowing the periodic events to be stored and queried by the database system. The temporal query language is introduced in the next chapter along with implementation of the model in O_2.

Chapter 5

Implementation

The implementation of the periodic temporal data model involves the following steps: (i) A complete implementation of temporal types instant, interval, element, and periodic elements as a abstract data type hierarchy. This step also involves the implementation of the set theoretic operators, temporal transformations, temporal comparison and other operators (The implementation currently supports absolute time only), (ii) Implementation of a periodic temporal extension to the O_2 data model by implementing temporal properties of objects as methods mapping time instants to attribute values, (iii) Implementation of valid time functions and methods for temporal attributes and objects, (iv) Implementation of periodic transaction time support.

Among other time related functions, we also provide temporal selection and temporal projection at the attribute level. Users can create and maintain a temporal database by first including the temporal type hierarchy and by providing the system with the schema definitions of the application from which the temporal schema is generated by the system. The system also comes with the following functions for computing various temporal constants:

- *zero()*: returns the beginning *instant* of the physical time.
- *infinity()*: used for specifying $\emptyset$.
- *Time()*: returns the whole time-line $[0, \infty)$ in the form of a periodic element.
- *future()*: returns the future $[now, \infty)$ in the form of a periodic element.
- *now()*: returns the current time instant.
- *until_now()*: returns the past $[0, now]$ in the form of a periodic element.

5.1 Temporal Types

The temporal types were defined as abstract data types in the previous chapter. Here we skip the implementation of aperiodic types instant, interval and element and only elaborate on the implementation of various operators on periodic elements. Note that since these types are defined by the **type** construct of O_2 model, the operations on the types are defined as **function**s.

```
type Duration :integer;
type Instant :integer;
type Interval :tuple (begin: integer,
                      end: integer);
type Element :list (Interval);
type P_element :tuple (aperiodicpart: Element,
                       periodicpart: Element,
                       period: Duration);
```

We prefix function names with type names to distinguish between different versions of overloaded function, because O_2 does not allow the overloading of functions. For example, the intersection between intervals and elements are computed by the *interval_intersect* and *element_intersect* functions respectively.

5.1.1 General Purpose Temporal Functions

In our approach temporal data is manipulated through a set of built-in functions defined on the aperiodic and periodic temporal types. These general purpose functions are called specialized functions in Chapter 2. Let *type* be one of the types of *Instant*, *Interval*, *Element*, and *P_element*. Then the following are the built-in temporal functions provided by the system:

- *begin (t: type): Instant* returns the starting point of event t.
- *end (t: type): Instant* returns the end point of event t.

- *read (s: string): type* converts the *s* string to a temporal value of *type*.
- *write (t: type): string* converts the temporal value of *type* to a string.
- *empty (t: type): Boolean* returns true if t is empty.
- *interval (i: Instant, j: Instant): Interval* creates an interval from instants i and j.
- *element (list (I: interval)): Element* creates a temporal element from the interval in the list.
- *p_element (aperiodicpart: Element, periodicpart: Element, p: Duration): P_element* creates a periodic element from two elements and a duration.
- *distance (i: type, j: type): Duration* computes the distance between two periodic event i and j.
- *length (i: type): Duration* computes the length of event *i*. The length of a periodic event may be infinite.
- *select_interval (i: type, k: integer): Interval* selects the *k*th interval from event *i*. *type* can be either *Element* or *P_element*.
- *first_interval (i: type): Interval* selects the first interval from event *i*. *type* can be either *Element* or *P_element*.
- *last_interval (i: type): interval* selects the last interval from event *i*. *type* can be either *Element* or *P_element*.

5.1.2 Set Theoretic Operators

Four different set theoretic operators union, intersect, difference, complement are defined for the *Interval*, *Element* and *Periodic element* (*P_element*) types. The set theoretic operators does not apply to *Instant* type. The function signature of these operators are given below.

- interval_union (i: Interval, j: Interval): Element
- interval_intersect (i: Interval, j: Interval): Interval
- interval_difference (i: Interval, j: Interval): Element
- interval_complement (i: interval): Element
- element_union (i: Element, j: Element): Element
- element_intersect (i: Element, j: Element): Element
- element_difference (i: Element, j: Element): Element
- element_complement (i: Element): Element
- union (i: P_element, j: P_element): P_element
- intersect (i: P_element, j: P_element): P_element
- difference (i: P_element, j: P_element): P_element
- complement (i: P_element): P_element

The O_2 query language OQL is a powerful query language and allows any expression as input parameters for a function. Thus it is possible to call set theoretic functions with OQL statements as parameters. The result type of a set theoretic function is the same as the type of its input parameters for *Element* and *P_element*. In the case of *Interval* type, the result can be of *Element* type (See the *interval_difference, interval_complement,* and *interval_union* functions above). Below we give the function body for the *union* and *intersect* functions. The method for computing the union and difference of periodic elements were in Section 0.

```
function body intersect (P: P_element, Q: P_element): P_element
{       o2 P_element R;
        o2 tuple (a: P_element, b: P_element) aux;

        if (empty (P) || empty (Q))
                return R;
        aux = synhronize (P, Q);
        R.aperiodicpart = element_intersect (aux.a.aperiodicpart, aux.b.aperiodicpart);
```

```
        R.periodicpart = element_intersect (aux.a.periodicpart, aux.b.periodicpart);
        R.period = aux.a.period == 0 ? aux.b.period : aux.a.period;
        return normalize (R);
};

function body uinon (P: P_element, Q: P_element): P_element
{       o2 P_element R;
        o2 tuple (a: P_element, b: P_element) aux;

        if (empty (P))
                return Q;
        if (empty (Q))
                return P;
        aux = synhronize (P, Q);
        R.aperiodicpart = element_union (aux.a.aperiodicpart, aux.b.aperiodicpart);
        R.periodicpart = element_union (aux.a.periodicpart, aux.b.periodicpart);
        R.period = aux.a.period == 0 ? aux.b.period : aux.a.period;
        return normalize (R);
};
```

The union and intersection algorithms are almost identical. The main difference is the operations (*element_union, element_intersect*) used for computing the periodic part and the aperiodic part of the result (*R*). The *synchronize* function called by both *union* and *intersect* synchronizes the aperiodic part, periodic part and period of two given periodic elements so that the *element_union* and *element_intersection* functions can be safely applied. The synchronization is obtained through the formatting functions *shiftright* and *unfold* as given in the algorithm below which is edited and simplified for clarity.

```
function body synchronize (P: P_element, Q: P_element):
                                        tuple(a: P_element, b: P_element)
{       o2 Duration period;
        o2 Instant d;
```

```
        d = instant_distance (       element_begin (P.periodicpart),
                                     element_begin (Q.periodicpart));
        if (instant_before ( element_begin (P.periodicpart),
                             element_begin (Q.periodicpart)))
                P = shiftright (P, d);
        else
                Q = shiftright (Q, d);
        if (P.period != Q.period) {
                period = lcm (P.period, Q.period);
                P = unfold (P, period / P.period);
                Q = unfold (Q, period / Q.period);
        return tuple (a: P, b: Q);
};
```

The *lcm* function calculates the least common multiplier of 2 integers.

5.1.3 Temporal Transformations

The temporal transformations were formally defined in Chapter 2. The aperiodic transformations functions are given below by their function signature:

- instant_translate (t: Instant, d: Duration): Instant
- interval_translate (I: Interval, d: Duration): Interval
- interval_scale (I: Interval, s: integer): interval
- element_translate (e: Element, d: Duration): Element
- element_scale (e: Element, s: integer): Element
- translate (P: P_element, d:Duration): P_element
- scale (P: P_element, s: integer): P_element

Note that *scale* function is not defined for *Instant* type. Below we give the algorithms for temporal transformations of periodic elements. The algorithms are written in O_2C and slightly edited and simplified for clarity. O_2C is an object-oriented extension of the C language.

```
function body translate (P: P_element, d: Duration): P_element
{ return tuple    (    aperiodicpart: elementtranslate (P.aperiodicpart, d),
                       periodicpart : elementtranslate (P.periodicpart, d),
                       period       : P.period);
};

function body scale(P: P_element, d: Duration): P_element
{ o2 P_element Q;
  o2 Instant     t;

if (empty (P))     return P;
Q.aperiodicpart = elementscale (P.aperiodicpart, d);
if (elementempty (P.aperiodicpart))
        t = 0;
else
        t = elementbegin (P.periodicpart) - elementbegin (P.aperiodicpart);
Q.periodicpart = elementtranslate (elementscale (P.periodicpart, d), t * d - t);
Q.period     = d * P.period;
return Q;
};
```

5.2 Temporal Objects

In Periodic Temporal Object-Oriented Model temporal objects and non-temporal objects can co-exist in a database. A temporal object can have temporal and non-temporal attributes. There has to be at least one temporal attribute in a temporal object, otherwise it is non-temporal.

The non-temporal attributes of a temporal object is defined as usual. The temporal attributes are defined as methods corresponding to temporal functions mapping time instants to attribute values. Let *attrib* be a temporal attribute name and

attrib_type be the type of the attribute. Let *Attrib* and *Attrib_type* be a non-temporal attribute and its type respectively. As a non-temporal object in class *class_name*, the definition would be

```
class class_name visibility type tuple
        Attrib: Attrib_type
        . . .
        {temporal} attrib: attrib_type
        . . .
end;
```

Then the corresponding temporal class is generated by the system given below (currently the temporal classes are generated manually.)

```
class class_name inherit t_object visibility type tuple
        Attrib: Attrib_type,
        . . .
        visibility t_attrib : list (tuple (time: p_element, value: attrib_type)),
        . . .
        transaction_time : P_element,
method
        /* attribute level temporal methods */
        visibility attrib (t: instant): attrib_type,
        visibility set_attrib (list (tuple (        time: p_element,
                                                    value: attrib_type))): boolean
        visibility display_attrib : boolean
        /* object level valid time methods */
        public vtime : p_element,
        public t_display: string
        /* object level transaction time methods */
        public is_active (t : Instant) : boolean
        public ttime : p_element,
        public set_ttime (P_element): boolean,
```

```
        public insert,
        public delete,
        public recall,
    end;
```

User provides the non-temporal class definition to the system with temporal attributes are indicated by the *temporal* keyword. The *temporal* keyword is only one way of specifying temporal attributes. Temporal attributes can also be specified in a different way. The generation of temporal class definition from non-temporal class definition involves the following steps:

5.2.1 Data Structure for Valid Time

For each temporal attribute *attrib*, the attribute definition *attrib: attrib_type* is replaced by a new attribute *t_attrib*. *t_attrib* is a data structure for storing the temporal assignment of the temporal attribute *attrib*. *t_attrib* is defined as a *list* of *tuple*s of *time* and *value* pairs. The *time* is specified as periodic element for periodic events. For applications involving strictly aperiodic events it could be of *Instant, Interval* or *Element*. The type of *value* attribute is the attribute domain *attrib_type*. The visibility of *t_attrib* is same as the visibility of the temporal attribute *attrib* with the exception of *public* visibility. The visibility of t_attrib can be either *private* or *read*. we do not allow arbitrary changes to the underlying data structure for maintaining temporal data. We allow *read* type visibility because this way users can write their own special purpose temporal functions. Thus the expressive power of our approach can be increased by defining new constructs in the form of functions.

5.2.2 Attribute Level Temporal Methods

For each temporal attribute a set of methods called *attribute level temporal methods* are generated by the system for accessing and modifying temporal data in a uniform fashion. These functions access, modifies and displays a temporal attribute. *(i)* ***visibility*** *attrib (t: instant): attrib_type* method computes the snapshot value of the *attrib* attribute at instant *t*. *t* could be any time within the lifespan of the attribute. *(ii)*

visibility *set_attrib (list (tuple (time: p_element, value: attrib_type))): boolean* function modifies the value over multiple durations of time in the form of a list. Each modification is specified by a tuple of *tuple (time: p_element, value: attrib_type)* type where *time* is the possibly periodic duration of time and *value* is the value during *time*. The previous *value*s during *time* is overridden by the *set_attrib* function. *(iii)* ***visibility*** *display_attrib : string* displays the temporal attribute attrib in the following format.

$time_1 \rightarrow value_1$,
$time_2 \rightarrow value_2$,
. . .
$time_n \rightarrow value_n$

5.2.3 Object Level Valid Time Methods

For every class a set of valid time methods are generated. (i) The ***public*** *vtime : p_element* method computes the valid time of an object from the valid times of its temporal attributes. (ii) The ***public*** *t_display: string* displays all the attributes of the object in the following format:

$attrib_1$
$time_1 \rightarrow value_1$,
$time_2 \rightarrow value_2$,
. . .
$time_n \rightarrow value_n$
$attrib_2$
$time_1 \rightarrow value_1$,
$time_2 \rightarrow value_2$,
. . .
$time_m \rightarrow value_m$
. . .

5.2.4 Transaction Time Support

Even though we are primarily concerned with valid time, we also provide transaction time support in the implementation of the periodic temporal data model. As with valid time, the use of periodic elements as time-stamps gives more power and flexibility to the model. In temporal databases, the transaction time is thought as aperiodic. The transaction time of objects is implicitly updated by the system as objects are inserted, deleted and recalled. We believe more can be achieved by empowering the user with the ability of modifying the transaction time explicitly, in addition to implicitly maintaining the transaction time by insertions, deletions and recalls. Consider seasonal employees who are hired year after year by a company. In a conventional transaction time database, the records of seasonal employees are manually *recall*ed at the beginning of each season and *delete*d at the end of each season every year periodically. In our approach such a transaction time can be represented by a periodic element eliminating the need for manual *recall*s and *delete*s. The transaction time related methods and data structures are again generated by the system which are discussed below.

Every class inherits the *transaction_time: P_element* attribute from the *t_object* class. *t_object* is the super class for all temporal classes. The *transaction_time* attribute stores the transaction time of temporal objects. Remember logical deletions/insertions reduce to physical modifications to the *transaction_time* of objects. The transaction time of a temporal object is accessed by the *ttime* method inherited from the *t_object* class. The transaction time of a temporal object can be modified by the ***public*** *set_ttime (t: P_element): boolean* method. The transaction time can be modified for the future time. Retrospective modifications i.e., modifications into the past are not allowed.

To be backward compatible with aperiodic transaction time we also support transaction time for *insert*, *delete*, and *recall* methods. The insert method creates a

temporal object, sets the transaction time to [now, ∞) initially and returns the object id of the new object. The *delete* method modifies the transaction time as follows

$$transaction_time \leftarrow transaction_time \cap [0, now]$$

Thus from *now* on the object is deleted logically. The *recall* method undoes the effect of the delete method as follows:

$$transaction_time \leftarrow transaction_time \cup [now, \infty)$$

meaning the object is inserted once more time at instant *now*. It is also possible to simulate the *delete* and *recall* methods by the *set_ttime* method. The ***public*** *is_active (t : Instant) : boolean* method checks if the object is a *current* or *active* object at instant t, i.e., $e \rightarrow is_active$ returns true for an object e if $t \cap e \rightarrow ttime \neq \emptyset$. thus it is possible to check if an object will be active (i.e. not logically deleted) at some future instant.

5.3 Temporal Queries

We introduce the periodic temporal queries in O_2 query language OQL. (Temporal databases can also be directly queried within an application through C, C++ and O_2C interfaces to O_2) The queries are expressed using the *Select* command similar to SQL Select. However, the Select command in O_2 is more powerful than the SQL Select. For example nested subqueries can be placed anywhere in the *select*, *from,* or *where* clauses as opposed to only in the *where* clause of SQL Select. The O_2 query processor is also capable of evaluating any valid expressions involving object-ids, mathematical formulas, function calls, etc. [O 95]

Consider the employee database in *Example 1.1*. We define an employee class for storing employee data where the birth date is a non-temporal attribute, name and salary are defined as temporal attributes. The name attribute could have been also defined as static attribute in which case the changes to employee names cannot be stored and queried.

```
class emp inherit t_object
public type
   tuple (birthdate: Date,
         t_empname: list (tuple (time: P_element, value: string)),
         t_salary: list (tuple (time: P_element, value: integer)),
         . . .)
method
         public empname (t: Instant): string,
         public set_empname (list (tuple (time:P_element, value: string))): boolean,
         public display_empname: string,
         public salary (t: Instant): string,
         public set_salary (list (tuple (time:P_element, value: string))): boolean,
         public display_salary: string,
         public vtime: P_element,
         public t_display (i: Instant): string,
         . . .
end;
```

We illustrate the periodic temporal query language through a set of queries on the employee class.

1. What is John's salary now?

```
select  e->salary (now())
from    e in emps
where   e->empname (now())=="john"
```

2. What was John's pay 2 weeks after he started working?

```
select  e->salary (instant_translate (begin (vtime_integer (e->t_salary)), 14))
from    e in emps
where   e->empname (now())="john"
```

The t_salary is the temporal extension of the salary attribute, i.e., it is a data structure containing the salary values over the past present and the future. The *vtime_attrib_type* (vtime_integer, vtime_string, etc) function computes the valid time of a temporal attribute of type *attrib_type*. The *begin (vtime_integer (e->t_salary))* computes the starting time of the salary attribute. The *instant_translate (begin (vtime_integer (e->t_salary)), 14)* translates the beginning time by 14 days.

3. How often is John paid ?

```
select  period (vtime_integer (e->t_salary))
from    e in emps
where   e->empname (now())="john"
```

vtime_integer (e->t_salary) computes the valid time (lifespan) of the salary attribute in the form of a periodic element. The *period* function returns the period of a periodic element.

4. Who are employees who are paid weekly?

```
select  e->empname (now())
from    e in emps
where   period (vtime_integer (e->t_salary)) = 7
```

5. How much do employees make as of today?

```
select  tuple (name: e->empname (now()), salary: e->salary(now()))
from    e in emps
```

This query return a set of tuples. The *tuple* construct creates a tuple from a list of attributes.

6. Display salary history of employees

    select *e->display_salary()*
    from *e in emps*

The *display_salary* attribute returns the salary over the lifespan of the attribute in text format.

7. Are all employees currently making more than $6 currently?

    for all *e in emps: e->salary(now()) >= 6*

A query can be existentially or universally quantified in OQL. This query evaluates to true if all employees make more than $6 currently.

8. Is there any employee making less than $5 now?

    exists *e in emps: e->salary (now ()) <= 6*

9. Will there be any employee making less than $6 in a year from now?

    exists *e in emps: e->salary (instant_translate (now(), 365)) <= 6*

The *instant_translate (now(), 365)* expression computes the temporal expression *a year from now* by translating current time instant (*now()*) by 365 days.

10. What is the maximum salary now?

    max (***select*** *e->salary (now())*
    from *e in emps)*

11. What was the maximum salary paid as of 9 days ago?

```
max (  select e->salary (instant_translate (now(), –9))
           from   e in emps)
```

12. List for all employees how many days they have been with the company until today?

```
select length (intersect (e->vtime(), until_now()))
from  e in emps
```

13. Who are those employees that have been in the company during the time John was in the company?

```
define john as
intersect (element (    select e->vtime()
                        from   e in emps
                        where  e->empname(now()) = "john"), until_now())

select f->empname(now())
from   f in emps
where  contain (f->vtime(), john)
```

The first part of the query computes the past portion of the lifespan of the object in the form of a periodic element and saves it as a temporary name (john). The *element* is an OQL function flattening a set of one value/object to a single value/object. The past portion of the lifespan of john object is computed by intersecting the past (*until_now()*) with the lifespan. The second part of the query returns all employees whose lifespan *contains* the lifespan of the *john* object. the same query can be expressed with a single OQL statement as follows

```
select f->empname(now())
```

```
from   f in emps
where  contain (f->vtime(), element (select e->vtime()
                                     from   e in emps
                                     where  e->empname(now()) = "john"))
```

13. Who are those employees worked in the company before John started working?

```
select f->empname(now())
from   f in emps
where  before (f->vtime(), element (select e->vtime()
                                    from   e in emps
                                    where  e->empname(now()) = "john"))
```

The subquery returns the time during which john was, is, and will be with the company, i.e., the lifespan. The lifespan of each object is returned by f->vtime().

14. What is the time in which either John or Mary are in the company?

```
Union (element (select e->vtime()
                from   e in emps
                where  e->empname(now()) = "john"),
       element (select e->vtime()
                from   e in emps
                where  e->empname(now()) = "tim"))
```

15. What is the number of employees now?

```
count (select   e
       from     e in emps
       where    e->is_active (now()))
```

16. What is the total number of employees in the company?

***count** (**select** e **from** e in emps)*

17. When does John get paid "greater than or equal to" $6 ?

select *pelementwrite (select_when (e->t_salary, “≥”, 6))*
from *e in emps*
where *e->empname(now())="john"*

The *select_when (t_attrib: list (tuple (time:P_element, value: attrib)), op: string, exp: attrib) : P_element* function performs a temporal selection based on a comparison. *op* can be one of “>”, “<”, “≤”, “≥”, “=”, “≠”. t_attrib is an expression of *t_attrib* type and *exp* is an expression of *attrib* (attribute domain) type.

Chapter 6

Modeling Calendars and Calendric Operations

A calendar is a means for specifying and reasoning about time used extensively in business applications. Events in temporal databases such as those in [CC 87, NA 87, G 88] are usually defined with respect to one or more calendar granularity. In general, calendars involve multiple granularity such as days, weeks, years. It is difficult to capture the semantics of calendars using aperiodic temporal constructs like instants, intervals [All 83], and temporal elements [GV 85]. Natural language expressions involving calendars (called *calendric expressions*) like *the first Monday of every month*, *the fourth Thursday of Novembers*, *the work days in year 1995*, *every year between Thanksgiving and Christmas* are even more difficult to represent and manipulate.

Calendars and calendric events are modeled in the literature using collections of intervals [L 86], linear repeating intervals [NS 92], temporal mediators [WJ 93], or object oriented types [Te 93]. A collection of intervals [L 86] is a hierarchically structured lists of intervals interpreted cyclically to define calendars and possibly infinite periodic events. Chandra et all [Ch 94] improved and implemented the calendar algebra in given [L 86]. This implementation supports periodic events within a finite interval of time. Michael Soo [So 93] proposes an extension to SQL2 [Me 90] that supports multiple calendars. The main idea of this proposal is to separate the user dependent features of calendars from the universal ones. The SQL2 extension provides the set of universal features for calendars, and leaves the user dependent aspects to the database managers. Temporal mediators [WJ 93] are a means for converting between different calendar granularity. The main purpose of this paradigm is to display the data in a temporal relational database in terms of a different time granularity. Generalized databases [KSW 90] uses linear repeating points or *lrp* (an infinite set of equally-spaced integers denoted by a linear formula) and a set of

constraints to model periodic events in relational databases. The use of *lrp*s gives rise to the duplication of data over multiple tuples. In [NS 92], Niezette and Stevenne propose linear repeating intervals (*lri*) instead of linear repeating points (*lrp*s) to overcome the difficulties of the *lrp*s in expressing calendars within the same theoretical framework. A *lpi* is a possibly infinite set of fixed length equally-spaced set of intervals. However some difficulty with generalized databases still exists. For example, a calendric event or just one calendar may need to be stored as a set of generalized tuples resulting in storage redundancy and compromised data integrity, because a calendar may correspond to more than one *lri* [NS 92]. An object-oriented approach is taken for modeling calendars in Eiffel language in [Te 93]. The implementation of calendars are based on instants (time points), intervals, and durations types in a similar fashion to [So 93].

The approaches mentioned above have a number of deficiencies: (*i*) The existing approaches don't distinguish between the *aperiodic* and *strictly-periodic* part of a *partially-periodic* event. With existing approaches such a temporal history/event has to be stored as at least two separate object resulting in storage redundancy, inefficient query processing. (*ii*) In most cases [WJ 93, Ch 94, So 93], the reasoning about periodic time is not fully automated, the user has to write a significant amount of code to describe how to do the conversions between calendars. (*iii*) Calendric events defined periodically with respect to other events like *every year between Thanksgiving and Christmas* can't be expressed by the existing approaches at symbolic level. (*iv*) A good calendar algebra should also support *and, or,* and *not* of natural languages. (*v*) Finally the existing approaches do not attempt to model relative calendric events. In this paper, we attempt to model calendars using periodic elements.

This chapter is organized as follows: The basic concepts are presented informally in Section 6.1. Specification and derivation of calendars from existing calendars are also presented in Section 6.2. Calendric events are discussed in 6.3. New calendar operators *periodic intervals*, *periodic interval selection* are introduced

Sections 0 and 0. In Section 6.6, the temporal relationships between periodic events are discussed. The modeling of relative events similar to absolute events is studied in Section 6.7. Calendric expressions are summarized in Section 6.8. Finally we summarize and conclude in Section 6.9.

6.1 Modeling Calendars

A periodic event is called a *basic calendar*, *time unit* or *granularity* (depicted in Figure 6.1) if it satisfies the following properties:

1. A basic calendar always start from *0*th clock tick.
2. Consecutive intervals in a basic calendar are abut in time, i.e., they neither overlap, nor there is a gap between two consecutive intervals.
3. A basic calendar consists of an infinite number of intervals, i.e., it extends into infinity.
4. The intervals are defined cyclically or periodically. The length of a duration cycle is called the period of calendar.

The period of the calendar of years is 4 years because of the leap years in which February is 29 days long. The most common examples for basic calendars are the calendar of seconds, the calendar of minutes, etc. Basic calendars are building blocks of more complex calendars and calendric events.

An *arbitrary calendar* (or calendar for short) is defined similar to basic calendars but differs from them in two aspects. (*i*) The intervals in a calendar need not be abut in time i.e. there can be gaps between 2 consecutive intervals. (*ii*) An arbitrary calendar may start from an instant other than the 0th clock tick. Examples of arbitrary calendars include the calendar of time-cards, calendar of fiscal year, the academic calendar, the calendar of paydays.

A calendric event is a temporal event expressed with respect to a calendar. Hence a calendric event can be aperiodic, strictly-periodic or partially-periodic event. Calendric expressions are formulas corresponding to the natural language expressions

on calendars in temporal applications. Calendric expressions are built from calendars and other calendric events through the temporal constructs extended for periodic time, the relational operators and the set theoretic operators.

We assume the existence of a smallest time interval called *chronon* that can be measured by hardware. A chronon is the time between 2 consecutive clock ticks of the system clock. For simplicity, we assume that the length of a chronon is a second.

Definition 6.1

A basic calendar α is a strictly-periodic event $(I_1, I_2, ..., I_n)\{p\}$ satisfying the following conditions for $n, p \in I$:

- $begin(I_1) = 0$.
- I_i *meets* I_{i+1} = *true* for $1 \leq i < n$.
- $p = \sum_{j=1}^{n} length(I_j)$

□

The formula $begin(\alpha) = 0$ enforces that the beginning of a basic calendar is the first clock tick. The formula I_i *meets* I_{i+1} = *true* assures that the intervals do not overlap and there is no gaps between consecutive intervals. The intervals $I_1, I_2, ..., I_n$ corresponds to the time units of the calendars. p is the period of the calendar. Basic and arbitrary calendars are depicted in Figure 6.1.

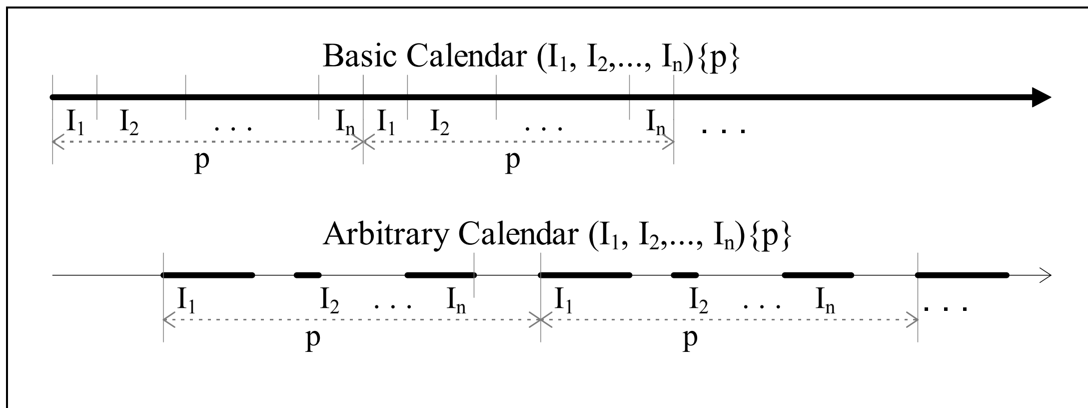

Figure 6.1: Basic and arbitrary calendars.

Example 6.1

The basic calendar of seconds (*Seconds*), minutes (*Minutes*), etc. can be given as a single periodic element in terms of clock ticks assuming the clock ticks at every second:

$$Seconds = [0, 1)\{1\} = [0, 1) \cup [1, 2) \cup [2, 3) \cup \ldots$$

$$Minutes = [0, 60)\{60\} = [0, 60) \cup [60, 120) \cup [120, 180) \cup \ldots$$

$$Hours = [0, 3600)\{3600\} = [0, 3600) \cup [3600,7200) \cup [7200,108000) \cup \ldots$$

□

Definition 6.2

An arbitrary calendar is a strictly-periodic event $(I_1, I_2, \ldots, I_n)\{p\}$ *where* I_i *before* I_{i+1} *= true for* $1 \leq i < n$.

□

The condition I_i *before* I_{i+1} = *true* states that there may be gaps between intervals in an arbitrary calendar. The intervals $I_1, I_2, \ldots, I_n$ need not be of equal length.

Example 6.2

The calendar of Mondays (*Mondays*), the calendar of 7-day work-hours 8:00am-12:00pm and 1:00pm-5:00pm (*Workhours*), the calendar of midnight's (*Midnights*) can be specified as follows:

$$Mondays = [0, 1d)\{7d\} = [0, 1d) \uplus [7d, 8d) \uplus [14d, 15d) \uplus \ldots$$
$$Workhours = ([8h, 12h], [13h, 17h])\{1d\} = [8h,12h] \uplus [13h,17h] \uplus [32h,36h] \uplus \ldots$$
$$Midnights = 0\ \{1d\} = 0 \uplus 1d \uplus 2d \uplus \ldots$$

where the constants h and d stands for 3600 and 86400 clock ticks respectively and used for brevity.

□

As seen from the examples above, specifying calendars in terms of the clock ticks (the physical time) is very cumbersome and difficult. This method may even be impractical for complex calendric events and for calendars like years and centuries. To overcome this difficulty we show how to derive calendars from existing calendars through temporal transformations in the next subsection.

6.2 Derivation of Calendars

Instead of explicitly specifying a calendar in terms of clock ticks, the mathematical relationships between calendars can be utilized for defining a calendar from existing ones in a straightforward manner. This can be done in two ways.

- Calendar C_1 can be a *scaled* version of calendar C_2,
- Calendar C_1 can be a *translation* of calendar C_2.

We think that using transformations is more natural and easier than using special constructs and operators just for the purpose of defining calendars as in [L 86, NS 92, Ch 94, So 92]. For example, the temporal transformations capture the semantics of *synchronization*, *duration* and *period* operators in [NS 92]. Since periodic elements

are closed under scaling and translation, every calendar evaluates to a single periodic element.

Example 6.3

Consider the calendar of minutes (*Minutes*) which is exactly the same as the calendars of seconds (*Seconds*) except the intervals are 60 times longer than the intervals in the calendar of seconds. Similar relationship exists between other calendars.

$$
\begin{aligned}
Minutes &= \underline{\underline{scale}}\ (Seconds, 60)\\
&= \underline{\underline{scale}}\ ([0, 1)\{1\}, 60)\\
&= [0, 60)\{60\}\\
&= [0, 60) \,\underline{\cup}\, [60, 120) \,\underline{\cup}\, \ldots\\
Hours &= \underline{\underline{scale}}\ (Minutes, 60)\\
&= [0, 3600)\{3600\}\\
&= [0, 3600) \,\underline{\cup}\, [3600, 7200) \,\underline{\cup} \ldots\\
Days &= \underline{\underline{scale}}\ (Hours, 24)\\
&= [0, 86400)\{86400\}\\
&= [0, 86400) \,\underline{\cup}\, [86400, 172800) \,\underline{\cup} \ldots\\
Weeks &= \underline{\underline{scale}}\ (Days, 7) = \ldots
\end{aligned}
$$

Consider the calendar of Mondays, Tuesdays, etc. Mathematically the calendar of Tuesdays is exactly the same as the calendar of Mondays except the intervals in it are shifted to the left by one day. Let $Mondays = [0, 1d)\{7d\}$ be the calendar of Mondays. Then

$$
\begin{aligned}
Tuesdays &= \underline{\underline{translate}}\ (Mondays, 1d)\\
&= [2d, 3d)\{7d\}\\
&= [2d, 3d) \,\underline{\cup}\, [9d, 10d) \,\underline{\cup}\, [16d, 17d) \,\underline{\cup} \ldots\\
Wednesdays &= \underline{\underline{translate}}\ (Mondays, 2d) = \ldots
\end{aligned}
$$

In a similar fashion, the calendar of months of equal-length can be derived from one another. For example, the calendar of months of length 31 days (January, March, May, July, etc.) can be derived from the calendar of March (*Marches*) as follows:

May = translate *(Marches, 61d)*

July = translate *(Marches, 122d)*

More complex relations between calendars can be expressed by a combination of temporal transformations. For example, the calendar of Aprils (30 days long) can be derived from the calendar of Marches (31 days long) as follows:

Aprils = translate *(*scale *(Marches, 30d/31d), 31d).*

□

We have studied the derivation of calendars above. Below we investigate calendric events and how they are specified in terms of a calendar.

6.3 Calendric Events

A periodic temporal event expressed with respect to a calendar is called a *calendric event*.

Definition 6.3 (Calendric event)

A calendric event α_C *is a temporal event* $\alpha = ([c_1, c_2], [c_3, c_4], ..., [c_5, c_6]):([c_7, c_8], [c_9, c_{10}], ..., [c_{n-1}, c_n])\{p\}$ *expressed with respect to the time units (intervals) of basic calendar* $C = [b, e)\{e-b\}$.

□

Here α_C serves as a mapping from C to *Chronons*, i.e., $\alpha_C = \underline{scale}(\alpha, e-b)_{Chronon}$. Each instant t in α corresponds to the beginning of the tth interval in C. The calendar of clock ticks is the default calendar, i.e., $\alpha = \alpha_{Chronon}$.

Example 6.4

The 5 work days (*Workdays*) can be specified in terms of the calendar of days (*Days*) as follows:

$$Workdays = [0, 5)\{7\}_{Days}.$$

where the numbers *0*, and *5* refers to the beginning points of the first and the 6th intervals in the calendar of days. The number *7* represents the length of *7* days. The 7 day work-hours (*Workhours*) can be given in the calendar of hours (*Hours*) as follows:

$$Workhours = ([8, 12],[13, 17])\{24\}_{Hours}.$$

The work days of an employee who works Mondays through Wednesdays after going through 2 full weeks of training may be represented by the following calendric event with respect to the calendar of days, assuming Monday is the first day in the calendar:

$$EmployeeHour = [0, 15): [15, 18)\{7\}_{Days}.$$

□

Since calendric events are represented by periodic elements, the set theoretic operations for periodic elements can be used on calendric events as illustrated by the example below.

Example 6.5

Weekends (*Weekends*) consists of Saturdays and Sundays:

$$Weekends = Saturdays \underline{\underline{\cup}} Sundays.$$

Workdays (*Workdays*) Monday through Friday, can be given by the difference or the union operator:

$$Workdays = Days \underline{\setminus} Weekends$$
$$= Mondays \underline{\underline{\cup}} \ldots \underline{\underline{\cup}} Fridays.$$

Workdays in the month of Januarys (*WorkdaysinJanuarys*) and in Jan. 1995 (*WorkdaysinJan95*) can be computed by the intersection operator where the results are strictly-periodic and aperiodic calendric events respectively.

$$WorkdaysinJanuarys = Workdays \mathbin{\underline{\cap}} Januarys.$$

$$WorkdaysinJan95 = Workdays \mathbin{\underline{\cap}} [jan\text{-}1\text{-}95, jan\text{-}31\text{-}95].$$

□

6.4 Periodic Interval Selection

Expressions like *every Christmas* (which may occur in an active database application), i.e., the set of all Christmases, can not be expressed easily, because Christmas is the 25th day of December which has a period of 4 years (because of leap years). In this section we introduce the periodic interval selection function on periodic elements to express such statements. Periodical interval selection is an extension of the interval selection function *select_interval* (see Section 2.3) on temporal elements to calendric events. The function *interval* selects an interval from a temporal element indicated by the parameter j, i.e., *select_interval* $((I_1, I_2, ..., I_n), j) = I_j$ for $1 \leq j \leq n$. Functions *first_interval*, *last_interval* are special cases of function *interval* selecting the first and the last intervals respectively. For $\alpha = ([3, 5], [7, 11])$, *select_interval*$(\alpha, 2) = [7, 11]$ and *first_interval* $(\alpha) = [3, 5]$. The periodic interval selection function *select_interval* selects intervals from calendric events periodically according to another calendric event or calendar. Periodical *first_interval* and *last_interval* functions are defined similarly.

Definition 6.4 (Periodic Interval Selection)

Let α and β be two calendric events. The periodical interval selection function $select_interval^{\beta}(\alpha, i)$ is a periodic application of function select_interval to event α within each interval in event β.

$$select_interval^{\beta}(\alpha, i) = \underline{\cup}_{k=1}^{\infty}\ \underline{select_interval}\ (\underline{select_interval}\ (\beta, k) \underline{\cap}\ \alpha, i).$$

□

The formula $\underline{select_interval}\ (\underline{select_interval}\ (\beta, k) \underline{\cap}\ \alpha, i)$ selects the *i*th interval from α within the duration of *k*th interval in β. Note that both events can be partially-periodic. The formula on the left handside eventually evaluates to a single periodic element because of the closure property of periodic elements under set theoretic operators. For aperiodic events, *select_interval* reduces to aperiodic interval selection function $\underline{select_interval}$.

The periodic interval selection is computed in a similar way the set theoretic operators are computed in Section 0. The computation of periodic interval selection is given below. In the procedure *app, pp* and *p* stand for *aperiodic_part, periodic_part* and *period* functions respectively. The computation proceeds in three steps: In the first step the beginning of the periodic part of the events are synchronized. In the second step, the periodic part of the events are synchronized, so that the period of both events are the same. In the third step, the select_interval operators is applied.

***Procedure**: select_interval (Computes the)*

***Input:**: α, β, i where α and β are periodic elements representing temporal events or calendars and i is an integer*

***Output**: $select_interval^{\beta}(\alpha, i)$*

1. *Shift the beginning of the periodic part so that that their app and pp are synchronized*
   if begin (pp (α)) before begin (pp (β))
   $\quad\alpha \leftarrow right\ (\alpha, begin\ (pp\ (\beta)) – begin\ (pp\ (\alpha)))$
   else
   $\quad\beta \leftarrow right\ (\beta, begin\ (pp\ (\alpha)) – begin\ (pp\ (\beta)))$
2. *Synchronize the period*
   if $p(\alpha)\ != p(\beta)$
   $\quad\alpha \leftarrow unfold\ (\alpha, lcm\ (p\ (\alpha), p\ (\beta))/p\ (\beta))$
   $\quad\beta \leftarrow unfold\ (\beta, lcm\ (p\ (\alpha), p\ (\beta))/p\ (\alpha))$

3. *Now apply the select_interval operator*
   $\gamma \leftarrow \cup_{k=1}^{m}$ $\underline{interval}$ $(\underline{interval}$ $(app(\beta), k) \cap app(\alpha), i) : \cup_{k=1}^{n}$ $\underline{interval}$ $(\underline{interval}$ $(pp(\beta), k) \cap pp(\alpha), i)$.

$m = \#ofinterval(app(\alpha)) = \#ofinterval(app(\beta))$
$n = \#ofinterval(pp(\alpha)) = \#ofinterval(pp(\beta))$

□

The following example illustrates the periodic interval selection.

Example 6.6

Let *Months* and *Years* be the calendar of months and years[1]. Then the calendar of Decembers (*Decembers*) can be specified by periodically selecting the 12th interval within every year:

$$Decembers = select_interval^{Years}(Months, 12).$$

The expression every Christmas (*Christmases*) can be specified by periodically selecting the 25 day of Decembers:

$$Christmases = select_interval^{Years}(Decembers \sqsubseteq Days, 25).$$

Note that the expression $select_interval^{Decembers}(Decembers \sqsubseteq Days, 25)$ also yields the same result. The last Friday of every January can be expressed by periodically choosing the last interval from the Fridays in Januarys (expressed by intersecting the calendar of Januarys with the calendar of Fridays).

$$LastFridayofJanuarys = last_interval^{Years}(Januarys \sqsubseteq Fridays).$$

□

---

[1] $Months = ([0, 31), ..., [1430, 1461))\{1461\}_{Days}$.

6.5 Periodic Intervals

Aperiodic (ordinary) intervals [Al 83] denote events with specific beginning and end times. It would be very desirable to express the statements like *every year between Thanksgiving and Christmas,* which might specify the time with the highest business activity in a business application, as a single interval at symbolic level as in *[Thanksgiving, Christmas]* with *Thanksgiving,* and *Christmas* representing the two periodic events involved. We introduce a new construct called *periodical interval* which is the natural extension of aperiodic interval construct to periodic elements:

Definition 6.5 (Periodic Interval)

Let $\alpha = N_\alpha{:}P_\alpha\{p\}$ and $\beta = N_\beta{:}P_\beta\{p\}$ be two calendric events with the same number of intervals in their aperiodic part N_α, N_β and periodic parts P_α, P_β. Then the periodical interval $\underline{[}\alpha, \beta\underline{]}$ is a calendric event whose *k*th interval is constructed by picking the beginning time from the *k*th interval in event α and the end time from the *k*th interval in event β:

$$\underline{[}\alpha, \beta\underline{]} = \underline{\cup}_{i=1}^{\infty} [begin\ (\underline{select_interval}\ (\alpha,i)),\ end\ (\underline{select_interval}\ (\beta,i))]$$

where *begin (select_interval (α,I))* ***beforeorequal*** *end (select_interval (β,i))* for $i > 0$.

□

The formula above evaluates to a single periodic element because of the fact that periodic elements are closed under set union. Hence periodic elements are closed under periodic intervals. Note that the *period* of α and β need not be a multiple of each other. Open ended periodic intervals are defined in similar fashion where open ends $\underline{(}$ and $\underline{)}$ of periodical intervals don't cover the corresponding intervals from events α and β. For instantaneous events $\alpha = t_1$ and $\beta = t_2$, the periodic interval $[\alpha, \beta]$ reduces to regular (aperiodic) interval $[t_1, t_2]$. For aperiodic events α and β where each event is a temporal element, the periodic interval $[\alpha, \beta]$ evaluates to another

temporal element. For example, for aperiodic events α = *[5, 9], [12, 16], [20, 25], [30, 35]* and β = *[7, 12], [13, 16], [23, 32], [31, 40]*, the periodic interval evaluates to *[α, β] = [5, 12], [12, 16], [20, 32], [30, 40]* which can be simplified to *[5, 16], [20, 40]*. Consider the following example for the application of periodic intervals.

Example 6.7

Let *Thanksgivings* = $select_interval^{Years}$ *(Novembers* $\underline{\cap}$ *Thursdays, 4)* and *Christmases* () be two calendric events denoting the set of all Thanksgivings (the fourth Thursday of Novembers) and Christmases. Then *[Thanksgiving, Christmas]* is a calendric event representing the time between Thanksgivings and Christmases every year including the day of Christmas and Thanksgiving.

□

The periodic interval are computed in a similar way the set theoretic operators are computed in Section 0. The computation of periodic intervals is given below. In the procedure *app, pp* and *p* stand for *aperiodic_part, periodic_part* and *period* functions respectively. The computation proceeds in three steps: In the first step the aperiodic part of the events are synchronized so that both events have the same number of intervals in their aperiodic part. In the second step, the periodic part of the events are synchronized, so that the period of both events are the same. In the third step, if both events have the same number of interval in their periodic part, then the periodic interval is constructed by picking the respective start end and times from the events. Otherwise periodic interval cannot be computed.

Procedure Periodic Interval: *Computes the periodic interval [α, β] from two given periodic elements α and β*

Input:*: Periodic elements α and β.*

Output*: Periodic interval [α, β]*

1 *Synchronize the aperiodic parts so that they have the same no of intervals*
if #ofinterval (app $(\alpha)) <$ *#ofinterval (app* $(\beta))$
for i = 1 to #ofinterval (app $(\beta))$ *– #ofinterval (app* $(\alpha))$
$\alpha \leftarrow$ *right* $(\alpha$, *begin* $(\underline{interval}\ (pp\ (\alpha), 2)) - begin\ (pp\ (\alpha)))$
else
for i = 1 to #ofinterval (app $(\alpha))$ *– #ofinterval (app* $(\beta))$
$\beta \leftarrow$ *right* $(\beta$, *begin* $(\underline{interval}\ (pp\ (\beta), 2)) - begin\ (pp\ (\beta)))$
2 *Synchronized the periodic part so that they have the same period*
if $p\ (\alpha)\ != p\ (\beta)$
$\alpha \leftarrow unfold\ (\alpha, lcm\ (p\ (\alpha), p\ (\beta))/p\ (\beta))$
$\beta \leftarrow unfold\ (\beta, lcm\ (p\ (\alpha), p\ (\beta))/p\ (\alpha))$
3 *If #ofinterval(app*$(\alpha))$ = *#ofinterval(app*$(\beta))$
$\gamma = \cup_{k=1}^{m}$ *[begin (interval (app* (α), *k)), end (interval (app* (β), *k))] :* $\cup_{k=1}^{n}$
[begin (interval (pp (α), *k)), end (interval (pp* (β), *k))].*
else
return nil.
m = *#ofinterval (app* $(\alpha))$ = *#ofinterval (app* $(\beta))$
n = *#ofinterval (pp* $(\alpha))$ = *#ofinterval (pp* $(\beta))$

□

We believe that using the familiar interval construct *[b, e]* to express events such as the above is a novel approach which can produce user-friendly query languages in which queries are expressed more naturally. To our knowledge only our approach has such a construct.

6.6 Temporal Relationships between Calendric Events

Temporal relationships between intervals are studied in detail [Al 83] and are adopted by temporal model for aperiodic data in [Sn 93] and others. Since we defined temporal relationships between periodic temporal events in Chapter 3 we don't go over the same concepts for calendric events, because calendric events are presented by periodic elements themselves.

6.7 Modeling Relative Events

Relative events are planned actions and designs for future such as the ordering of tasks in a production line, the presentation order of a set of multimedia streams, or a set of synchronized transactions in a real-time system. Relative events (*temporal templates*) can be expressed similarly to absolute events using a special kind of periodic element called *relative periodic element*. Relative events are defined with respect to a *virtual reference point* (vrp), i.e., the unknown beginning of the event denoted by the • symbol. Since we use periodic elements to represent and manipulate calendars and calendric events, it is possible to defined relative calendars and calendric events though relative periodic element (See 3.2). Here we demonstrate relative periodic calendric events by some examples.

Example 6.8

A week is a duration of 7 days:

$$[\bullet, 7)\{7\}_{Days}$$

The duration of a task performed in 3 parts can be given as a *relative temporal element*:

$$([\bullet, 5], [10, 15], [20, 30]).$$

A multimedia presentation consists of synchronized multimedia streams such as audio, video, graphics etc. The play out of video sequences with 24 and 12 frames per second with a duration of 1/48 seconds can be stated (assuming the clock ticks 48 times a second for simplicity) as

$$\alpha = [\bullet, 1)\{2\} \text{ and } \beta = [\bullet, 1)\{4\}.$$

□

Temporal transformation becomes necessary when planned events are to be aligned with respect to one another. We present these operators informally here since they are defined similar to absolute transformations.

Example 6.9

Consider the video streams above. If the user wants to start the second video stream (β) 10 seconds (480 ticks) after the start of the first stream, this could be done by translating the beginning of β by 10 second with respect to vrp of the presentation.

$$translate(\beta, 10) = (\bullet, \bullet) : [480, 481)\{4\}.$$

□

Note that $(\bullet, \bullet)$, or simply $\bullet$, serves as a reference point only and is not part of the event. If sequences are to be played faster or slower, the events should be scaled. To slow down the play out of the first video stream, the user can scale up the event by a factor of two:

$$scale\ (\alpha, 2) = [\bullet, 2)\{4\}.$$

We illustrate the relative-to-absolute conversion using the multimedia sequences above (The event in a multimedia presentation can be stated with respect to different calendars such as the calendar of 24 frames per second for video, the calendar of 12 frames for graphics, etc.)

Example 6.10

The actual presentation of the video sequences α and β at instant 100 results in the following absolute multimedia events α' and β'

$$\alpha' = anchor([\bullet, 1)\{2\}, 100) = [100, 101)\{2\}.$$

$$\beta' = anchor([\bullet, 2)\{4\}, 100) = [100, 102)\{4\}.$$

□

6.8 Calendric Expressions

Calendric expressions are built from calendric events and calendric operators including *the set theoretic operators* $\cup$, $\cap$, \, ~, *the periodical interval operators* [ ], ( ], [ ), ( ), and *the periodical interval selection operator select_interval*. Calendric expressions can be defined recursively as follows:

1. *Choronon = [0, 1){1}* (the physical time, i.e., clock ticks) is a calendric expression.
2. Every basic or arbitrary calendar C is a calendric expression.
3. Every calendric event α, or α_C is a calendric expression (If C is omitted in α_C it is assumed to be *Chronon*).
4. Let α and β be calendric expressions, *C* and *D* basic calendars. Then $\alpha_C \cap \beta_D$, $\alpha_C \cup \beta_D$, $\alpha_C \setminus \beta_D$, $\sim \alpha_C$, $select_interval^{\alpha}(\beta_D, i)$, $[\alpha_C, \beta_D]$, $(\alpha_C, \beta_D]$, $[\alpha_C, \beta_D)$, (α_C, β_D) are calendric expressions.

Optimization is an important part of query evaluation in temporal query languages. Calendric expressions can be optimized for efficient query evaluation. Optimization can be performed at two different places. (i) Choosing a set of simplification rules for a given expression (ii) choosing a calendar to perform an operation between two events stated with respect to different calendars. These issues are discussed below.

6.8.1 Calendar Conversion

When events α_C and β_D in basic calendars *C* and *D* are involved in a calendric expression like $\alpha_C \cap \beta_D$, a conversion of α_C and β_D to a another calendar *E* must be performed first, otherwise the intersection will produce an incorrect result. The easiest way to do the intersection is to convert the events α_C, and β_D to *Chronon* calendar first, then to perform the intersection. The result can then be converted to a desired calendar. However a more efficient evaluation can be obtained by choosing a

calendar *E* whose period is the greatest common divisor (*gad*) of the periods of *C* and *D*, that is,

period (E) = gad (period(C), period (D)).

The *period* function was defined in Section 3.1.2. This rule is called the *calendar optimization rule*. For instance, for calendars *Weeks* and *Days*, the rule chooses *Days* as the calendar to perform operations, because *period (Days) = gcd (period (Days), period (Weeks))*.

6.9 Summary

We treated calendars and calendric events as special types of periodic events. Calendric events are periodic events expressed with respect to a calendar. We use the temporal transformations, set theoretic operators, and temporal relational operators of periodic elements to express calendric events, which makes the model simple and user-friendly. Relative calendric events are defined similarly. Our calendar algebra on periodic elements are closed for absolute and relative periodic events with conversions between the two. We extended the interval construct [b, e], called *periodic interval,* for calendric events to express complex calendric statements. The *periodical interval selection* function selects intervals from periodic calendric events according to another event. To our knowledge these two constructs are supported only in our model (the selection operators in [L 86, Ch 94, NS 92] operates only on strictly-periodic or aperiodic events).

Chapter 7

Related Research and Comparison

[So 93, Te 93, WJ 93] mainly deal with aperiodic calendric events. We briefly compare our work with TSQL2 first, then with collection of intervals [Le 86, Ch 94], generalized databases [KSW 90, NS 92] which exclusively deal with periodic time. Our approach differs from the previous related work since it (i) represents partially-periodic events as a single temporal value, (ii) models relative events as well as absolute events uniformly, and (iii) expresses events involving periodic intervals at symbolic level using the periodic interval construct.

7.1 TSQL2

TSQL2 [S 95] is a temporal query language based on relational model for querying and manipulating time-varying data. It is upward-compatible with the international standard query language SQL-92. Some of the main characteristics of TSQL2 can be summarized as follows: TSQL2 support only one valid-time dimension. The model is based on tuple versioning or tuple time stamping. TSQL2 is based on homogeneous tuples., i.e., the attributes in a tuple have the same valid time. The valid time support includes both the past and the future. Time stamps are not limited in range at least in the design of the language. The user-defined time support includes instants, intervals and durations. Temporal support is optional at the table level. It is possible to define non-temporal tables. Multiple calendars support and some calendar related operations are supported. Temporal aggregates are supported in TSQL2.

TSQL2 is meant to be a consensus temporal database model and language for relational model. The model support aperiodic temporal events. Event though the language supports multiple calendars, calendar granularity conversions and similar operators, the underlying time stamps are instants intervals, temporal elements and durations. There is not a support for strictly-periodic or partially periodic events in the

language design. The relative time is supported only by durations. There is no periodic relative time support.

7.2 Collection of Intervals

A collection of intervals [Le 86] is a hierarchical list structure to represent events. For example, the collection of months where each month is represented by a collection of the days in that month in order is an order 2 collection. Suppose *Weeks* is a calendar of the weeks in 1993.

Weeks = {(-4, 3), (4, 10), (11, 17), (18, 24), (25, 31), (32, 38), (39, 45), ...}

Two general types of operators are defined on collections: *slicing* and *dicing*. The *strict dicing* takes a collection *C*, an interval *t* and a relational interval operator *R* such as overlaps, during, etc. and breaks up t into pieces according to C.

$$C{:}R{:}t \equiv \{c \sqcap t \mid c \in C \text{ and } c\ R\ t = true\}$$

For example, *Weeks:overlap:<January-93>* breaks *the <January-93>* interval on the week boundaries, i.e., it will give the whole and partial weeks during *<January-93>* interval. Assume that *<January-93>* is represented by interval *(1, 31)*. Then

{Weeks:overlap:<January-93>} = {(1, 3), (4, 10), (11, 17), (18, 24), (25, 31)}

The *selection* operators, *f/C* and $[f_1, f_2, ..., f_n]/C$, applies the selection functions *f* (which could be an integer or *n* or *–n*) to the collection *C* and returns the corresponding intervals or collections from C. *n/C* and *–n/C* select the *n*th interval from the beginning and end of *C* respectively. Collection of intervals are implemented in [Ch 94]. A set of examples on calendar expressions in collection of calendars and periodic elements are given in Table 7.1. A question mark indicates that we could not express the given statement in the formalism.

Table 7.1: Collection of intervals versus periodic elements.

Event	*Collection of intervals*	*Periodic Elements*
Mondays	2/Days:during:Weeks	select_interval^{Weeks}(Days, 2)
Januarys	1/Months:during:Years	first_interval^{Years}(Months)
First day of every month	1/Days:during:Months	first_interval^{Months}(Months $\cap$ Days)
First Monday in January 86	1/Mondays:during:Januarys: during:1986/Years	first_interval (Mondays $\cap$ Januarys) $\cap$ select_interval(Years, 1986))
Every year between Thanksgiving and Christmas	?	[Thanksgiving, Christmas]

7.3 Generalized Relations

In [KSW 90] a framework for handling infinite temporal data based on relational data model is proposed. Infinite temporal data is represented by what is called a *generalized tuple* consisting of a set of temporal attributes, a set of time stamps, and a set of constraints in the form of inequalities. In this framework two time stamps are used to represent a possibly infinite set of intervals, one for the beginning and the other for the end time of intervals. Each time stamp is called a *linear repeating point* (lrp). A lrp is the set of time instants computed by the formula $a+kn$ where a and k are integer constants and n takes integer values between $-\infty$ and ∞. For example $2+3n$ corresponds to the following set of instants $\{..., -4, -1, 2, 5, 8, ...\}$. Then two *lrp*s can represent a possibly infinite set of intervals. Thus generalized tuples can be thought as a tuple with a time stamp consisting of a possibly infinite number of intervals. As argued by many authors tuple time stamping is more redundant, has less modeling power than attribute time stamping.

In this framework the use of *lrp*s and constraints may obscure the time represented by a time stamp. It may not clear to the user what an infinite set of

intervals a pair of lrps and a set of constraints represent. It should also be noted that the set of intervals represented by a pair of *lrp*s are not mutually disjoint as illustrated by the examples below.

Example 7.1

The generalized tuple [1, 1+2n], $X_2 \geq 0$ represent the finite set of tuples {[1, 1],[1, 3],[1, 5], ...}. However one can easily see that this tuple can be represented by the following interval [1, ∞), because the intervals are not pair-wise disjoint.

□

Example 7.1

The generalized tuple [$3+2n_1$, $5+2n_2$], $X_1=X_2 - 2$ represent the finite set of tuples {..., [1, 3], [3, 5], [5, 7] ...}. Similarly one can easily see that this tuple can be represented by the following interval (-∞, ∞) which is the union of the intervals denoted by the generalized tuple.

□

Example 7.1

The generalized tuple $[1+2n_1, 3n_2 - 4], X_1 = X_2 - 2$ represent the finite set of tuples *{..., [1, -4], [3, -1], [5, 2], [7, 5], [8, 8], [11, 11], [13, 14], [15, 17] ...}*. As seen in this example some of the intervals e.g. *[5, 2]* generated by the generalized tuple are in disagreement with the definition of interval and must be disregarded. Therefore every generalized tuple should include the constraint $X_1 \leq X_2$ to eliminate these invalid intervals.

□

One limitation of this framework is that the aperiodic and periodic part of an event are representable only in separate generalized tuples as illustrated by the below example. However it is undesirable to split data into multiple tuple because of the restrictions of the underlying model.

Example 7.1

Let us say that $robot_1$ performs $task_3$ in the following set of intervals {[1, 5], [10, 12], [14, 16], [18, 20], [22, 24], ...}. Note the interval [1, 5] is different in length and position than the rest of the intervals which can be represented by ($robot_1$, $task_3$, [$2n_1$, 2+n2] $X_1=X_2-1$, $X_1 \geq 10$). As a result we have two generalized tuples

($robot_1$, $task_3$, [$2n_1$, $2+n_2$] {$X_1=X_2-1$, $X_1 \geq 10$})
($robot_1$, $task_3$, [1, 5], {})

The first generalized tuple can be rewritten in an infinite number of different ways, for example ($robot_1$, $task_3$, [n_1, n_2] $X_1+1=X_2$, $X_2 \geq 12$). Given a set of intervals it is not clear how to get a minimal number of generalize tuple with the set of minimal number constraints and the simplest lprs in each tuple.

□

With every generalized tuple representing an infinite number of intervals, there must always be a set of constraint unless a single interval namely $(-\infty, \infty)$ is being represented, because it can be shown that an arbitrary pair of lrps represent the whole domain of time i.e. $(-\infty, \infty)$. (Rational: For every $x \in (-\infty, \infty)$ there is a pair number n_1 and n_2, where $X_1=a_1+k_1n_1$, $X_2= a_2+k_2n_2$, such that $x \in [a_1+k_1n_1, a_2+k_2n_2]$)

7.4 Calendars and Slices

The generalized tuples are defined for relational model using tuple time stamping. The use of *lrp*s and constraints makes it difficult to use in a real world application.

Niezette and Stevenne [NS 92] try to overcome this shortcoming by using *linear repeating intervals (lri)* in place of *lrp*s. A *lri* is given by the following formula: $kn+(b, e)$ where k, b, and e are integers. b and e denotes the beginning and end time of fixed length intervals.

Every *lrp* and *lri* can be given as single periodic element as follows: $a+kn \equiv a\{k\}$ and $kn+(b, e) \equiv (b, e)\{k\}$. (The $\equiv$ symbol states that the formulas on both sides of the symbol are equivalent. The set of constraints can be translated to set operators on periodic elements.) However the opposite in not true. For example, periodic element $[t_1, t_2], [t_3, t_4], \ldots [t_{k-1}, t_k] : [t_{k+1}, t_{k+2}], \ldots [t_{n-1}, t_n]\{p\}$ with each interval is of distinct length can be represented only by a set of *lpi*s (because an *lpi* are a single repeating fixed-length interval):

$n+[t_1, t_2]$

. . .

$n+[t_{k-1}, t_k]$

$np+[t_{k+1}, t_{k+2}]$

. . .

$np+[t_{n-1}, t_n]$

A calendar may corresponds to more than one generalized tuples each with one *lri* [NS 92] causing data duplication. Authors propose the *slice* $P \rhd D$ operator to choose intervals from calendars where $P = O_1C_1+ \ldots +O_nC_n$ is the starting points of intervals from a set of calendars $C_1, \ldots, C_n$, and $D = N.C$ is a duration of N intervals in calendar C. We informally compare slices and periodic elements in Table 7.2.

Table 7.2: Slices versus periodic elements.

Event	***Slices***	***Periodic Elements***
Sundays	weeks+1.days	$\text{select_interval}^{\text{Weeks}}(\text{Days}, 1)$
The last day of every month	?	$\text{last_interval}^{\text{Months}}(\text{Days})$
The 3rd day of every month	months+3.days	$x = \text{select_interval}^{\text{Months}}(\text{Months} \cap \text{Days}, 3)$
The 4th hour of the 3rd day of each month	months+3.days+4.hours	$\text{select_interval}^{\text{Days}}(x \cap \text{Hours}, 4)$
Every year between Thanksgiving and Christmas	?	[Thanksgiving, Christmas]

7.5 Summary

We compared our approach with related research in this Chapter. The related research concentrates on two points. Calendars and temporal databases. Generalized databases extend relational model with constraints on linear repeating points. A generalized tuple consists of a set of attributes and two *lrp*s. Generalized tuples are inherently more complex than other representations of periodic time. A generalized tuple can represent only one linear repeating interval. However in a real world temporal events such as scheduling requires more powerful representation. For example the schedule of a part time employee who comes to work twice a week can be represented by only two generalized tuples. Generalized tuples are based on tuple versioning. It is not clear whether this approach can be applied to attribute versioning at all. Event though linear repeating intervals remedies some of the limitations of generalized tuples, they still suffer the complexity of constraints associated with them. We compared the slice operator of [NS 92] to periodic elements. There is no construct in [NS 92] corresponding to our periodic interval.

Collection of interval is based on temporal elements. A collection of interval can represent a strictly-periodic event but not a partially-periodic one. The *dicing* and

slicing operators are powerful operators for representing calendar related events. Again there is no construct in [Le 86, Ch 94] corresponding to our periodic interval.

Chapter 8

Conclusion

Periodic temporal events occur in a variety of application domains such as calendars, scheduling, active, multimedia, scientific databases. Traditional temporal databases can store aperiodic events only and are incapable of reasoning about periodic data. Periodic data is stored in files not in a database and manipulated in an ad hoc manner. The existing proposals for modeling periodic time has a number of drawbacks such as inability to represent partially-periodic events (resulting in data duplication), lack of support for relative events (resulting in ad hoc methods), the use of mathematical formulas (compromising user-friendliness), incompatibility with the existing aperiodic types (preventing seamless extension of aperiodic models), lack of closure under set theoretic operators etc. The contributions of this thesis can be summarized as follows:

1. *Modeling of periodic time through a new type called periodic element and its algebra*. It has been recognized that existing temporal periodic types are inadequate for handling periodic events which come up in today's complex temporal applications. The aim of this part of the research is to represent and reason about different types of periodic events through a common data type called *periodic element* which provides user with a set of powerful operators such as set theoretic operators, relational operators, temporal transformations. This type and its operators also establish a formal framework on top of which more complex temporal applications can be built as illustrated in the second and the third part of the research. The significance of periodic element over the existing periodic temporal types is that it can represent absolute/relative, aperiodic, strictly-periodic, and partially-periodic events in a concise and straightforward manner.

2. *Modeling relative time*: Event though there are many applications requiring relative time, the issue of representing and reasoning about relative time has not been addressed properly in the literature. In spite of the fact that there are many similarities between absolute and relative time, the two kinds of time are not the same. We proposed relative aperiodic types *relative intervals, relative temporal element* for modeling relative aperiodic events and *relative periodic element* for modeling relative partially-periodic events. The behavior of these types are again defined via the set theoretic operators, the temporal transformations and the temporal relationship operators with clear semantics.

3. *Extending O_2 data model for periodic temporal events and its implementation*. Scheduling and planning applications in medicine, accounting, banking, law etc. can be easily developed, maintained, and queried by extending temporal databases with periodic elements. We introduced periodic time into object-oriented O_2 by modeling temporal attributes as a function of time and temporal types (instant, interval, element, and periodic element) as abstract data types (ADT). The significance of making time a function is that the temporal and non-temporal data are treated uniformly. By modeling time as a function and periodic elements as an ADT, O_2 is extended without changing the model or the language. Thus the temporal and non-temporal data are treated uniformly. Time is associated with data values instead of objects to avoid data duplication. This periodic temporal extension is implemented to demonstrate the feasibility of our approach in the O_2C language. O_2C is an object oriented language obtained by extending the C language with object-oriented database programming concepts. This implementation can be extended to include calendars and calendric events.

4. *Modeling Calendars and Calendric Events*: Temporal events are usually expressed with respect to a calendar which is a partitioning of the physical time-line. Calendric events occur frequently in business applications where an event may involve multiple calendars, such as the first Monday of April. By describing calendars through periodic elements, complex calendars and calendric events can

be expressed easily. We believe by making calendars a part of the temporal database system, we can relieve the user of the work of maintaining calendars in an ad hoc manner. By making calendars a special type of periodic event we eliminate the need for a separate data model. We extended the interval construct [b, e] for calendric events to express complex calendric statements. The periodical interval selection function selects intervals from periodic calendric events according to another event. To our knowledge these two constructs are supported only in our model (the selection operators in [Le 86, Ch 94, NS 92] operate only on strictly-periodic or aperiodic events).

The periodic events occur very frequently in real world, however they have not been studied extensively in the context of temporal databases. Many issues concerning the periodic temporal databases needs to be studied such as temporal joins, query optimization, index structures etc. Calendar expressions can be expressed using the set theoretic operator on periodic elements. However more specialized operators may need to be defined to express more complex expressions. Periodic transaction time is a new concept. The valid time corresponding to each interval of the transaction time of an event may be considered separately or the valid time can be considered as a whole. These and other issues remain as future work.

Bibliography

[A 93] Ilsoo Ahn. *SQL+T: a Temporal Query Language*. *Proceedings of the ARPA/NFS International Workshop on an Infrastructure for Temporal Databases.* Arlington Texas June 14-16, 1993.

[AM 89] Abadi M., Manna Z. *Temporal Logic Programming*. *Journal of Symbolic Computation*, pp. 277-295, 1989.

[All 83] Allen, J. F. *Maintaining knowledge about temporal intervals*. *Comm. of ACM*. 26 (11), Nov. 1983, pp. 832-843.

[All 84] Allen, J. F. *Toward a general Model of Action and Time*. *Artificial Intelligent* 23(2) 1984.

[B 93] Blakeley J. A., *Challenges for Research on Temporal Databases*. *Proceedings of the ARPA/NFS International Workshop on an Infrastructure for Temporal Databases* Arlington Texas June 14-16, 1993.

[BP 85] Barbic, F. and Pernici, B. *Time Modeling in Office Information Systems*. *Proc. of ACM-SIGMOD* 1985, pp. 51-62.

[B 93] Baudinet M., Chomicky J., Wolper P. *Temporal Databases Beyond: Beyond Finite Extensions*. *Proceedings of the ARPA/NFS International Workshop on an Infrastructure for Temporal Databases.* Arlington, Texas, June 14-16, 1993.

[B 91] Baudinet M., Niezette M., and Wolper P., *On the representation of infinite temporal Data and Queries*. *ACM SIGART_SIGMOD_SIGART Symposium on the Principles of Database Systems*, pp. 280-290, Denver CO, May 1991.

[Ba 92] Bancilhon, F., Delobel, C., and Kanellakis, P., "Building an Object Oriented Database System: The Story of O2", Morgan Kaufmann, 1992.

[Ca 94] Chandra R., Segev A., Stonebraker M. *Implementing Temporal Rules in the Next Generation Databases*. Conference on Data Engineering, 1994.

[CC 87] Clifford J. and Crocker A. *The Historical relational data model and algebra based on lifespans*. *Proceedings of the third international conference on Data Eng*, pp. 528-537, Los Angeles, Feb 1887.

[Cat 94] Cattell R.G.G., "The Object Database Standard: ODMG-93", Edited by Rick Cattell, Morgan Kaufmann, 1994.

[Ch 93] Chandra, Segev A., Stonebraker M. *Implementing Calendars and Temporal Rules in Next Generation Databases*., *Proceedings of the ARPA/NFS International Workshop on an Infrastructure for Temporal Databases.* Arlington Texas June 14-16, 1993.

[CI 88] Chomicki J., Imielinsky T. *Temporal Deductive Databases and Infinite Objects. ACM Pods* 1988.

[Ch 94] Chomicki J, *Temporal Query Languages: A Survey,* Temporal Logic, First International Conference ICTL 94, Bonn, Germany, July 94 Proceeding, Lecture Notes in Artificial Intelligent, 827. pp 480-534.

[CT 85] Clifford, C. and Tansel, A. U. *On an Algebra for historical relational databases: Two Views*, in Proceedings of ACM SIGMOD International Conference on Management of Data Austin Texas USA May 1985 pp. 247-265.

[DW 93] Umeshwar D. and Wuu G., *Extending DBMSs to Manage Temporal Data: An Object Oriented Approach*, *Proceedings of the ARPA/NFS International Workshop on an Infrastructure for Temporal Databases.* Arlington Texas June 14-16, 1993.

[EW 90] Elmasri R. and Wuu G., *A temporal Model and Query Language for ER Databases*. *Proceedings of the 6th International Conference on Data Engineering*, Feb. 1990, pp. 76-83.

[GB 93] Gadia S., Bhargava. *SQL-like seamless query of temporal data*. *Proceedings of the ARPA/NFS Int. Workshop on an Infrastructure for Temporal Databases.* Arlington Texas June 14-16, 1993.

[G 88] Gadia S. *Homogeneous Relation Model and Query Languages for Temporal Databases*. ACM Transactions of Databases, 1988 Dec. 13(4).

[GB 89] Gabbay D., and Mcbrien P. *Temporal Logic and Historical Database*. *Proc of 17th Conference on VLDB.*

[GV 85] Gadia, S. and Vaishav J. H.. *A query Language for Homogeneous Temporal Database*. in Proceedings of the ACM Symposium on Principles of Database Systems, pp. 51-56 March 1985.

[I 87] Imielinsky, I. *Intelligent Query Answering in Rule based Systems*. *Journal of Logic Programming*. 4(2), 1987.

[Je 94] Christian J. *A Consensus Glossary of Temporal Database Concepts*. *SIGMOD Record*, 23(1), March 1994, pp. 52-64.

[KO 95] Kurt, A. and Ozsoyoglu, M. *Modeling Periodic Time and Calendars*, in Proceedings of Second International Conference on the Applications of Databases, San Jose CA, 1995.

[KO 95] Kurt, A. and Ozsoyoglu, M. *Modeling and Querying Periodic Temporal Databases*, in Database and Expert Systems Applications, 6th International Conference London, United Kingdom, September 1995, Workshop Proceedings DEXA 95, pp. 124-133.

[L 86] Leban B., McDonald D. D., and Forster D. R. A *Representation for Collections of Temporal Intervals*. *AAAI* 1986 pp. 367-371.

[LG 93] Little T. and Ghafoor A., *Interval based Conceptual Models*. *IEEE Transactions on Knowledge Engineering*. Aug. 93.

[LBG 93] Ted Lawson, Carmel Balthazaar, Alex Gray. *An object-Oriented Approach to Temporal Modeling*. *Proceedings of the ARPA/NFS Int. Workshop on an Infrastructure for Temporal Databases.* Arlington Texas June 14-16, 1993.

[NA 87] Navathe S. B. and Ahmed R. *TSQL-A language interface for history database*. *Proceeding of The Conference on The Temporal Aspects in Information Systems*, pp. 113-128 France May 87. AFCET Noth-Holland.

[KSW 90] Kabanza F., Stevenne J. M., Wolper P., *Handling Infinite temporal data. in Ninth Annual ACM SIGACT-SIGMOD-SIGART* Symposium on Principles of Database Systems, pp. 392-403, Nashville TN. April 1990.

[Me 94] Melton J. *ISO/ANSI Working Draft SQL (SQL3),* August 29, 1994.

[NS 82] Niezette M., Stevenne J. M., *An Efficient Symbolic Representation of Periodic Time*, CIKM 1992, pp 161-168.

[O 95] The O_2 Object-Oriented Database System, OQL User Manual, 1995.

[RS 91] Rose E., and A. Segev. *TOODM - A temporal Object-oriented Data Model with Temporal Constraints*. in *Proceedings of the 10th International Conference on the Entity Relationship Approach*. Oct. 1991.

[S 94] Richard Snodgrass, et al. *TSQL2 Language Specification*. *SIGMOD Record*, 23(1), March 1994, pp. 65-86.

[S 94b] Snodgrass R. et al. *A TSQL2 Tutorial*, SIGMOD RECORD, Vol. 23, No. 1, March 1994.

[SC 91] Su, S.Y.W. and H. M. Chen *A temporal Knowledge Representation Model OSQM*/T and its Query Language OQL/T. Proceedings of the Conference on Very Large Databases*, 1991.

[Sn 87] Snodgrass R. *The temporal Query Language Tquel,* ACM Transactions on Database Systems, 12(2):247-298, July 1987.

[So 93] Soo M.. *Multiple Calendar Support for Conventional Database Management* Systems. *Proceedings of the ARPA/NFS International Workshop on an Infrastructure for Temporal Databases.* Arlington Texas June 14-16, 1993.

[Sn 94] Snodgrass R. *Temporal Object-Oriented Databases: A Critical Comparison*. Technical Report 1994.

[SS 87] Segev A., and Shosani A. *Logical Modeling of Temporal Data.* in Proceedings of ACM SIGMOD Annual Conference on Management of Data, pp. 454-466, San Francisco, CA, May 1987.

[S 91] Sciore, E. *Using Annotations to Support Multiple Kinds of Versioning in an Object-oriented Database System*. *ACM Transactions on Database Systems*, 16, No. 3, Sep. 1991, pp. 417-438.

[S 95] The TSQL2 Temporal Query Language, edited by Richard T. Snodgrass, Kluwer International Series in Engineering and Computer Science, Kluwer academic Publishers. 1995

[T 86] Tansel, A. *Adding time Dimension to Relational Model and Extending relational Algebra*. *Information Systems.* 11(4) 1986.

[T 91] Tansel A. U. *A Historical Database*. *Information Sciences*, No 53 (1991), pp. 101-133.

[T 93] Tansel A. U., *SQL^T: A temporal extension to SQL*. *Proceedings of the ARPA/NFS Int Workshop on an Infrastructure for Temporal Databases* Arlington Texas June 14-16, 1993.

[Ta 93] Tansel A. U. Clifford J. Gadia S. Jajodia S. Segev A. Snoddgrass R., *Temporal Databases: Theory, Design and Implementation*. The Benjamin Cummings Publishing Company 1993.

[Te 93] Lawson T., Bathhazaar C., Gray A., *An Object-Oriented Approach to Temporal Modeling, Proceedings of the ARPA/NFS Int. Workshop on an Infrastructure for Temporal Databases.* Arlington Texas June 14-16, 1993.

[W 92] Worboys M. F. A *model for spatio-temporal modeling. Proceedings of the 5th Int. Conference. Symposium on Spatial Data Handling*. Aug. 3-7 1992 South Carolina USA.

[WD 92] Wuu, G. and U. Dayal. *A Uniform Model for Temporal Object-oriented Databases*. *Proceedings of the International Conference on Data Engineering*. Tampa, Arizona, Feb. 1992, pp. 584-593.

[W 93] Wiederhold G. *Need to harmonize Temporal Data Model*. *Proceedings of the ARPA/NFS International Workshop on an Infrastructure for Temporal Databases*. Arlington Texas June 14-16, 1993.

[WJ 93] Wang, X. S., Jajodia, S., Subrahmanian, V. S., *Temporal Modules: An Approach Toward Federated Temporal Databases*, *Proc of the 1993 SIGMOD*, Washington DC, May 1993.

9. Appendix

9.1 Types

```
type Duration :integer;
type Instant :integer;
type Interval :tuple(begin: Instant,
                end: Instant);
type Element :list(Interval);
type P_element :tuple(aperiodicpart: Element,
                periodicpart: Element,
                period: Duration);
```

9.2 Class Definitions

```
class t_object inherit Object private type
   tuple(transaction_time: P_element)
method
   public isactive(t: Instant): boolean,
   public ttime: P_element,
   public delete,
   public recall,
   public insert: t_object
end;

class emp inherit t_object
rename
   method isactive from class t_object as active
public type
   tuple(t_empno: T_integer,
        t_salary: T_integer,
method
   public empname(t: Instant): string,
   public salary(t: Instant): integer,
   public set_empname(ta: T_string): boolean,
   public t_display(i: Instant): string,
   public set_salary(ta: T_integer): boolean,
   public display_salary: string,
   public display_empname: string,
   public insert: t_object
end;
```

9.2.1 Methods

```
method body isactive in class t_object
```

```
{   return overlap (p_element(list(interval(now(), now())), list(), 0), self-
>transaction_time); };

method body ttime in class t_object
{   return self->transaction_time; };

method body delete in class t_object
{   self->transaction_time = intersect (self->transaction_time, until_now()); };

method body recall in class t_object
{   self->transaction_time = Union (self->transaction_time, future()); };

method body insert in class t_object
{   o2 t_object p = new t_object;

   self->transaction_time = p_element(list(),
      list(interval(now(), instanttranslate(now(), 1))), 1);
   return self;
};

method body empname in class emp
{   o2 tuple(time:P_element, value:string) tv;

   for (tv in self->t_empname)
      if (overlap (p_element(list(interval(t, t)), list(), 0), tv.time))
         return (tv.value);
   return (o2 string) "";
};

method body empno in class emp
{   o2 tuple(time:P_element, value:integer) tv;

   for (tv in self->t_empno)
      if (overlap (p_element(list(interval(t, t)), list(), 0), tv.time))
         return (tv.value);
   return (o2 integer) 0;
};

method body salary in class emp
{   o2 tuple(time:P_element, value:integer) tv;

   for (tv in self->t_salary)
      if (overlap (p_element(list(interval(t, t)), list(), 0), tv.time))
         return (tv.value);
```

```
   return (o2 integer) 0;
};

method body vtime in class emp
{   o2 P_element vtime;
   o2 tuple(time:P_element, value:dept) tv;

   /* for all the attributes in class compute the vtime, take a union */
   vtime = Union (vtime, vtime_string (self->t_empname));
   vtime = Union (vtime, vtime_integer (self->t_salary));
   return vtime;
};

method body set_empname in class emp
{   o2 tuple (time: P_element, value: string) tv1, tv2;
   /* value can not be changed retrospectively            */
   /* i.e. the time of values in t_a must be later than now */

   if (self->t_empname != list()) { /*attribute*/
      for (tv1 in self->t_empname) /*attribute*/
         for (tv2 in ta) {
            if (tv1.value == tv2.value)
               tv1.time = Union (tv2.time, intersect (tv1.time,
                  p_element(list(interval (zero(), begin(tv2.time))), list(), 0)));
            if (overlap (tv1.time, tv2.time) && (tv1.value != tv2.value))
               tv1.time = difference (tv2.time, tv1.time);
         };
   } else
      /* add new temporal assignment */
      self->t_empname = ta; /*attribute*/
   return true;
};

method body t_display in class emp
{   return self->display_empname + "\n" + self->display_salary;
};

method body set_salary in class emp
{   o2 tuple(time:P_element, value:integer) tv1, tv2;

   for (tv1 in ta)
      for (tv2 in ta)
         if ((tv1 != tv2) && overlap (tv1.time, tv2.time))
            return false;
```

```
    self->t_salary += ta; /* incomplete */
    return true;
};

method body display_salary in class emp
{   o2 string S = "Salary:(";
    o2 tuple(time:P_element, value:integer) tv;

    for (tv in self->t_salary)
        S += pelementwrite (tv.time) + "->" + instantwrite (tv.value) + ", " ;
    return S + ")\n";
};

method body display_empname in class emp
{   o2 string S = "Name: ";
    o2 tuple(time:P_element, value:string) tv;

    for (tv in self->t_empname)
        S += pelementwrite (tv.time) + "->" + tv.value +  "\n";
    return S + ".";
};

method body insert in class emp
{   o2 emp p = new emp;

    self->transaction_time = p_element(list(),
        list(interval(now(), instanttranslate(now(), 1))), 1);
    return self;
};
```

9.3 Instant

9.3.1 Specialized Functions

```
function body instant(t: Time): Instant
{   return t; };

function body instantdistance(i: Instant,
                              j: Instant): Duration
{ return ((i > j)?(i - j):(j - i)); };

function body instantread(S: string): Instant
{   o2 Instant t = 0;
    o2 integer i, j, aux;

    if (S == ""){
```

```
        perror ("Instant specification error!");
        return 0;
    };
    for ((t = j = 0, i = count(S)-1); i >= 0; (j++, i--)) {
        aux = ((o2 integer) S[i] - 48);
        if (aux > 9 || aux < 0)
            perror ("Unable to convert instant!");
        t += aux * power (10, j);
    };
    return (t);
};

function body instantwrite(i: Instant): string
{   o2 string S = "";
    char c;
    do {
        c = (i % 10) + '0';
         S = " " + S;
        S[0] = c;
    } while ((i /= 10) > 0);
    return S;
};
```

9.3.2 Temporal Transformations

```
function body instanttranslate(t: Instant,
                               d: Duration): Instant

{   return (t + d); };
```

9.3.3 Temporal Comparisons

```
function body instantafter(i: Instant,
                           j: Instant): boolean
{ return (i > j); };

function body instantafterorequal(i: Instant,
                                  j: Instant): boolean
{   return (instantafter (i, j) || instantequal (i, j)); };

function body instantbefore(i: Instant,
                            j: Instant): boolean
{ return (i < j); };

function body instantbeforeorequal(i: Instant,
```

```
                          j: Instant): boolean
{   return (instantbefore (i, j) || instantequal (i, j)); };

function body instantequal(i: Instant,
                     j: Instant): boolean
{ return (i == j); };
```

9.4 Interval

9.4.1 Specialized Functions

```
function body intervalcount(P: P_element): integer
{   if (elementempty (P.periodicpart))
      return count (P.aperiodicpart);
   else
      return infinity ();
};

function body interval(begin: Instant,
                   end: Instant): Interval
{   o2 Interval I;
   if (instantafter (begin, end)) {
      perror("Unable to create interval");
      I.begin    = end;
      I.end     = begin;
   } else {
      I.begin = begin;
      I.end    = end;
   }
   return (I);
};

function body intervaldistance(I: Interval,
                         J: Interval): Duration
{   if (intervaloverlap(I, J))
      return 0;
   else
      if (intervalbefore(I, J))
         return (instantdistance (I.end, J.begin));
      else
         return (instantdistance (I.begin, J.end));
};

function body intervalisinstant(I: Interval): boolean
{   return instantequal(I.begin, I.end); };
```

```
function body intervalwrite(I: Interval): string
{       return "[" + instantwrite (I.begin) + ", " + instantwrite (I.end) + "]";
};

function body intervallength(I: Interval): Duration
{    return instantdistance (I.end, I.begin); };

function body intervalread(S: string): Interval
{    o2 Interval I;

    while ((' ' in S) >= 0)
        S[(' ' in S):(' ' in S)] = "";
    if (count (S) == 0) {
        perror ("No interval specified!");
        return I;
    };
    if ((('[' in S) >= 0) && ((']' in S) >= 0) && ((',' in S) >= 0)) {
        I.begin = instantread (S [1 : (',' in S)-1]);
        I.end   = instantread (S [(',' in S)+1 : (']' in S)-1]);
    } else if ((('[' in S) < 0) && ((']' in S) < 0)  && ((',' in S) < 0))
        I.end = I.begin = instantread (S);
    else
        perror ("Interval specification error!");
    return I;
};
```

9.4.2 Temporal Transformations

```
function body intervalscale(P: Interval,
                        d: Duration): Interval
{    return interval (P.begin, instanttranslate (P.begin, (P.end-P.begin)*d)); };

function body intervaltranslate(I: Interval,
                            d: Duration): Interval
{    return interval (instanttranslate (I.begin, d), instanttranslate (I.end, d)); };
```

9.4.3 Temporal Comparisons

```
function body intervalmeet(I: Interval,
                       J: Interval): boolean
{ return (instantequal (I.end, J.begin)); };

function body intervalmetby(I: Interval,
                        J: Interval): boolean
```

```
{ return (instantequal (I.begin, J.end)); };

function body intervaloverlap(I: Interval,
                              J: Interval): boolean
{ return
     ((instantbeforeorequal (J.begin, I.begin) && instantbeforeorequal (I.begin, J.end))
   || (instantbeforeorequal (J.begin, I.end)     && instantbeforeorequal (I.end, J.end))
   || (instantbeforeorequal (I.begin, J.begin) && instantbeforeorequal (J.begin, I.end))
   || (instantbeforeorequal (I.begin, J.end)     && instantbeforeorequal (J.end, I.end)));
};

function body intervalafter(I: Interval,
                            J: Interval): boolean
{ return (instantafter (I.begin, J.end)); };

function body intervalbefore(I: Interval,
                             J: Interval): boolean
{ return (instantbefore (I.end, J.begin)); };

function body intervalcontain(I: Interval,
                              J: Interval): boolean
{ return (instantbeforeorequal (I.begin, J.begin) && instantafterorequal (I.end,
J.end)); };

function body intervalequal(I: Interval,
                            J: Interval): boolean
{ return (instantequal (I.begin, J.begin) && instantequal (I.end, J.end)); };

function body intervalfinish(I: Interval,
                             J: Interval): boolean
{ return (instantequal (I.end, J.end)); };
```

9.4.4 Set Theoretic Operators

```
function body intervalunion(I: Interval,
                            J: Interval): Interval
{   /* assuming I overlaps J */
   return (interval (instantbefore (I.begin, J.begin) ? (I.begin) : (J.begin),
                instantafter  (I.end, J.end)     ? (I.end)   : (J.end)));
};

function body intervalintersect(I: Interval,
                                J: Interval): Interval
{   return (interval ((instantafter (I.begin, J.begin) ? I.begin : J.begin),
```

```
              (instantbefore (I.end, J.end) ? I.end : J.end))); };

function body intervaldifference(I: Interval,
                        J: Interval): Element
{   o2 Element P;
   if (intervaloverlap (I, J)) {
      P = list ();
      if (instantbefore (I.begin, J.begin))
         P += list (interval (I.begin, J.begin));
      if (instantafter (I.end, J.end))
         P += list (interval (J.end, I.end));
      return (elementnormalize (P));
   }else
      return (list (I));
};

function body intervalcomplement(I: Interval): Element
{   o2 Element P;
   if ((I.begin == zero()) && (I.end == infinity()))
      P = list ();
   if ((I.begin == zero()) && (I.end != infinity()))
      P = list (interval (I.end, infinity()));
   if ((I.begin != zero()) && (I.end != infinity()))
      P = list (interval (zero(), I.begin), interval (I.end, infinity()));
   return (P);
};
```

9.5 Element

9.5.1 Specialized Functions

```
function body elementwrite(P: Element): string
{   o2 string S = "";
   o2 Interval I;

   for  (I in P)
      S += intervalwrite (I);
   return S;
};

function body empty(P: P_element): Instant
{   return elementempty (P.aperiodicpart) && elementempty (P.periodicpart); };

function body end(P: P_element): Instant
{   if (empty(P)) {
```

```
        perror ("Unable to return the end of p element!");
        return (infinity ());
    };
    if ( elementempty(P.periodicpart))
        return intervallast(P.aperiodicpart).end;
    else
        return infinity ();
};

function body elementintervalcount(P: Element): integer
{ return (count (P));};

function body elementlength(P: Element): Duration
{   if (elementempty (P))   return (0);
    return instantdistance (intervallast (P).end, intervalfirst (P).begin);
};

function body elementnormalize(P: Element): Element
{   o2 Element Q;
    o2 Interval I;
    o2 integer j;
    if (count (P) < 2)   return P;
    Q = P[0:0];
    j = 0;
    for (I in P[1:])
        if (intervaloverlap (Q[j], I))
            Q[j] = intervalunion (Q[j], I);
         else {
            Q += list (I);
            j++;
        };
    return Q;
};

function body elementread(S: string): Element
{   o2 Element P;
    o2 string aux;
    o2 integer i;

    while ((' ' in S) >= 0)
        S[(' ' in S):(' ' in S)] = "";
    if ((S == "") || (S[count(S)-1] != ']')) {
        perror ("Unable to read element!");
        return P;
```

```
    };
    aux = S;
    i = (']' in aux);
    while (i >= 0) {
        P  += list (intervalread (aux[0:i]));
        if (i == (count (aux)-1))
            break;
        else
            if (aux[i+1] == ',') {
                aux = aux [i+2 : count(aux)-1];
                i = (']' in aux);
            } else {
                perror("Element specification error!");
                return P;
            }
    }
    return P;
};

function body elementsort(P: Element): Element
{   o2 Interval I, J;
    o2 integer j;
    o2 Element Q = list();
    for (I in P) {
        j = 0;
        for (J in Q where (instantbefore (J.begin, I.begin)))
            j++;
        if (j < count (Q))
            Q[j:j] = list (I) + Q[j:j];
        else
            Q += list (I);
    };
    return (Q);
};
```

9.5.2 Temporal Transformations

```
function body elementscale(P: Element,
                           d: Duration): Element
{   o2 Element Q;
    o2 Interval I;
    o2 Duration q;

    if (elementempty (P))
```

```
      return P;
   if (d < 1) {
      perror ("Unable to scale by a factor of zero!");
      return Q;
   };
   q = instantdistance (elementbegin (P), zero ());
   for (I in elementtranslate (P, -q))
      Q += list (interval (I.begin * d, I.end * d));
   return elementtranslate (Q, q);
};

function body elementtranslate(P: Element,
                        d: Duration): Element
{   o2 Element     Q;
   o2 Interval    I;

   for (I in P)
      Q += list (intervaltranslate (I, d));
   return Q;
};
```

9.5.3 Temporal Comparisons

```
function body elementmeet(P: Element,
                     Q: Element): boolean
{   if (elementempty (P) || elementempty (Q))
      return false;
   return (intervalmeet (intervallast(P), intervalfirst(Q)));
};

function body elementmetby(P: Element,
                     Q: Element): boolean
{   if (elementempty (P) || elementempty (Q))
      return false;
   return (intervalmeet (intervallast(Q), intervalfirst(P)));
};

function body elementoverlap(P: Element,
                       Q: Element): boolean
{   o2 Interval I, J;
   o2 boolean ok = false;
   for (I in P)
      for (J in Q)
         if (intervaloverlap (I, J))
```

```
            return (true);
   return (false);
};

function body elementoverlappedby(P: Element,
                          Q: Element): boolean
{ return (elementoverlap (P, Q)); };
function body elementstart(P: Element,
                     Q: Element): boolean
{    if (elementempty (P) || elementempty (Q))
       return false;
   return (intervalstart (intervalfirst (P), intervalfirst (Q)));
};

function body elementstartedby(P: Element,
                        Q: Element): boolean
{ return (elementstart (P, Q)); };

function body equal(P: P_element,
               Q: P_element): boolean
{    o2 tuple (a:P_element, b:P_element) R =  synhronize (P, Q);

   return (elementequal (R.a.aperiodicpart, R.b.aperiodicpart) &&
         elementequal (R.a.periodicpart,  R.b.periodicpart));
};

function body finish(P: P_element,
               Q: P_element): boolean
{    if (empty (P) || empty (Q))
       return false;
   return (instantequal (end (P), end (Q)));
};

function body finishedby(P: P_element,
                  Q: P_element): boolean
{ return finish (P, Q); };

function body future: P_element
{ return p_element(list(), list(interval(now(), instanttranslate(now(), 1))), 1);};

function body infinity: Instant
{    return (9999); };
```

9.5.4 Set theoretic Operators

```
function body elementdifference(P: Element,
                                Q: Element): Element
{   return (elementintersect (elementnormalize (P), elementcomplement
(elementnormalize (Q)))); };

function body elementintersect(P: Element,
                               Q: Element): Element
{   o2 Interval I, J;
   o2 Element T = list ();
   o2 Element R = elementnormalize (P);
   o2 Element S = elementnormalize (Q);
   for (I in R)
      for (J in S where (intervaloverlap (I, J)))
         T += list (intervalintersect (I, J));
   return (T);
};

function body elementunion(P: Element,
                           Q: Element): Element
{   return (elementnormalize (elementsort (P + Q))); };
```

9.6 Periodic Element

9.6.1 Specialized Functions

```
function body intervallast(P: Element): Interval
{   if (P == list ())
      perror ("Unable to get the last interval of element!");
   else
      return (P[count(P)-1]);
};

function body intervalfirst(P: Element): Interval
{   if (P == list ()) {
      perror ("Unable to get first interval from element!");
      exit (0);
   };
   return (P[0]);
};

function body intervalget(P: Element,
                          i: integer): Interval
{   if ((i < 1) || (i > count(P))) {
```

```
      perror("Subscipt out of bounds!");
      exit (0);
   };
   return (P[i-1]);
};

function body normalize(P: P_element): P_element
{   o2 integer i;

   P = reduce (P);
   for (i = elementintervalcount (P.periodicpart) / 2; i = 2; i++)
      if ((P.period % i == 0) && (elementintervalcount (P.periodicpart) % i == 0)) {
         P = fold (P, i);
         break;
      };
   while (! elementempty (P.aperiodicpart))
      if (shiftabletoleft (P, instantdistance (intervallast (P.aperiodicpart).begin,
elementbegin(P.periodicpart))))
         P = shiftleft (P, instantdistance (intervallast (P.aperiodicpart).begin,
elementbegin(P.periodicpart)));
      else
         break;
   return P;
};

function body aperiodicpart(P: P_element): Element
{ return (P.aperiodicpart); };

function body p_element(app: Element,
                  pp: Element,
                  p: Duration): P_element

{   o2 P_element P;
   if (((elementbefore (app, pp) || elementmeet (app, pp))
   && (elementlength (pp) <= p)) || (elementempty (app) || elementempty (pp))) {
      P.aperiodicpart    = app;
      P.periodicpart     = pp;
      P.period           = p;
      return P;
   } else {
      perror ("Unable to create periodic element");
      return P;
   };
};
```

```
function body shiftleft(P: P_element,
                  d: Duration): P_element
{
   o2 P_element    R;
   o2 integer       i, k;
   o2 Instant        t;

   if ((d < 0) || elementempty (P.periodicpart) || elementempty (P.aperiodicpart)) {
      perror ("Unable to shift left! Negative value.");
      return P;
   };
   R.aperiodicpart    = elementintersect (P.aperiodicpart, list(tuple(begin:zero(),
end:??)));

   return R; /* normalize (R); */
};

function body shiftright(P: P_element,
                   d: Duration): P_element
{
   o2 P_element    R;
   o2 integer       i, k;
   o2 Instant        t;

   if (d < 0) {
      perror ("Unable to shift right! Negative value.");
      return P;
   };
   if (elementempty (P.periodicpart))
      return P;
   R.aperiodicpart    = P.aperiodicpart;
   R.period        = P.period;
   k = d / P.period;
   for (i = 1; i <= k; i++)
      R.aperiodicpart += elementtranslate (P.periodicpart, (i-1)*P.period);
   t = elementbegin (P.periodicpart);
   R.aperiodicpart += elementintersect (elementtranslate (P.periodicpart, k*P.period),
list (interval(t, t+d)));
   R.periodicpart = elementintersect (elementunion (elementtranslate (P.periodicpart,
k*P.period),      elementtranslate (P.periodicpart, (k+1)*P.period)),
                list (interval (t+d, t+d+P.period)));
   return reduce (R);
};
```

```
function body synhronize(P: P_element,
                    Q: P_element): tuple(a: P_element,
                                          b: P_element)
/*  synhronizes periodic elements P and Q, so that they have the    */
/*  period and their periodic parts start approximately from the    */
/*  same instant                                                    */
{   o2 Duration period;
    o2 Instant d;

    if (elementempty (P.periodicpart) && elementempty (Q.periodicpart))
        return tuple (a: P, b: Q);
    else if (elementempty (P.periodicpart) && !elementempty (Q.periodicpart)
            && instantafter (elementend (P.aperiodicpart), elementbegin
(Q.periodicpart)))
        Q = shiftright (Q, instantdistance (elementend (P.aperiodicpart), elementbegin
(Q.periodicpart)));
    else if (! elementempty (P.periodicpart) && elementempty (Q.periodicpart)
            && instantafter (elementend (Q.aperiodicpart), elementbegin
(P.periodicpart)))
        P = shiftright (P, instantdistance (elementend (Q.aperiodicpart), elementbegin
(P.periodicpart)));
    else if (! elementempty (P.periodicpart) && !elementempty (Q.periodicpart)) {
        d = instantdistance (elementbegin (P.periodicpart),
                    elementbegin (Q.periodicpart));
        if (instantbefore ( elementbegin (P.periodicpart),
                elementbegin (Q.periodicpart)))
            P = shiftright (P, d);
        else
            if (instantbefore ( elementbegin (Q.periodicpart),
                    elementbegin (P.periodicpart)))
                Q = shiftright (Q, d);
        if (P.period != Q.period) {
            period = lcm (P.period, Q.period);
            P = unfold (P, period / P.period);
            Q = unfold (Q, period / Q.period);
        };
    };
    return tuple (a: P, b: Q);
};

function body unfold(P: P_element,
                n: integer): P_element
{   o2 P_element    R;
```

```
    o2 integer       i;

    if (n < 1)
        perror ("Unable to unfold! Negative value.");
    if (empty (P))
        return R;
    R.aperiodicpart    = P.aperiodicpart;
    if (elementempty (P.periodicpart))
        return R;
    R.periodicpart    = P.periodicpart;
    for (i=2; i <= n; i++)
        R.periodicpart += elementtranslate (P.periodicpart, (i-1) * P.period);
    R.period = P.period * n;
    return R;
};

function body begin(P: P_element): Instant
{   if (empty(P)) {
        perror ("Unable to return the beginning of p element!");
        return (zero ());
    };
    if (P.aperiodicpart == list ())
        return (P.periodicpart)[0].begin;
    else
        return (P.aperiodicpart)[0].begin;
};

function body distance(P: P_element,
                       Q: P_element): Duration
{   if (empty (P) || empty (Q)) {
        perror ("Unable to determine distance.");
        return 0;
    };
    if (overlap (P, Q))
        return 0;
    if (before (P, Q))
        return instantdistance (end (P), begin (Q));
    if (before (Q, P))
        return instantdistance (end (Q), begin (P));
    return 0;
};

function body pelementread(S: string): P_element
```

```
{   o2 P_element P;
    o2 string app, pp, p;
    o2 integer aux, aux1;

    while ((' ' in S) >= 0)
        S[(' ' in S):(' ' in S)] = "";
    if (S == "") {
        perror ("Periodic element specification error!");
        return P;
    }
    aux = (':' in S);
    if (aux >= 0) {
        aux1 = ('{' in S);
        if ( aux1 < 0 ) {
            perror ("Periodic element specification error! 2");
            return P;
        }
        app = S[0 : aux-1];
        pp  = S[aux+1 : aux1-1];
        p   = S[aux1+1 : count(S)-2];
    } else {
        aux = ('{' in S);
        if (aux >= 0) {
            app = "";
            pp  = S[0 : aux-1];
            p   = S[aux+1 : count(S)-2];
        } else {
            app = S;
            pp  = "";
            p   = "";
        };
    };
    if (app != "")
        P.aperiodicpart = elementread (app);
    if (pp != "")
        P.periodicpart  = elementread (pp);
    if (p != "")
        P.period        = durationread (p);
/*  check if the input consistent with the defintion of per elem */
    return (P);
};

function body length(P: P_element): Duration
{   if (empty (P))   return 0;
```

```
   if ((elementempty (P.periodicpart)) || (P.period == 0))
      return elementlength (P.aperiodicpart);
   else
      return infinity ();
};

function body pelementwrite(P: P_element): string
{   o2 string S = "";

   if (elementempty (P.aperiodicpart))
      if (elementempty (P.periodicpart))
         ;
      else
         if ((elementintervalcount (P.periodicpart)==1) &&
            (intervallength (P.periodicpart[0]) == P.period))
            S += "["+ durationwrite (P.periodicpart[0].begin)+", ->)";
         else
            S += elementwrite (P.periodicpart)
               + "{" + durationwrite (P.period) + "}";
   else
      if (elementempty (P.periodicpart))
         S += elementwrite (P.aperiodicpart);
      else {
         S += elementwrite (P.aperiodicpart) +":";
         if ((elementintervalcount (P.periodicpart)==1) &&
            (intervallength (P.periodicpart[0]) == P.period))
            S += "["+ durationwrite (P.periodicpart[0].begin)+", ->)";
         else
            S += elementwrite (P.periodicpart)
               + "{" + durationwrite (P.period) + "}";

      }
   return S;
};

function body period(P: P_element): Duration
{   return (P.period); };

function body periodicpart(P: P_element): Element
{  return (P.periodicpart); };

function body reduce(R: P_element): P_element
{
   if (empty(R))
```

```
      return R;
   R.aperiodicpart = elementnormalize (R.aperiodicpart);
   R.periodicpart  = elementnormalize (R.periodicpart);
   return R;
};

function body fold(P: P_element,
               n: integer): P_element
{   o2 P_element    R;
   o2 integer       i, m;

   if (n < 1) {
      perror ("Unable to unfold! Negative value.");
      return R;
   };
   if (empty (P))
      return R;
   R.aperiodicpart    = P.aperiodicpart;
   if (elementempty (P.periodicpart))
      return R;
   m = count(P.periodicpart);
   if (((m % n) != 0) || (m < n) || (P.period % n) != 0) {
      perror ("Unable to perform fold. Factor n not divisible!");
      return R;
   };
   R.periodicpart    = (P.periodicpart)[0 : (m/n)-1];
   R.period = P.period / n;
   for (i=2; i <= n; i++)
      if (! elementequal (R.periodicpart, (P.periodicpart)[(i-1)*m/n : (i*m/n)-1]))
         perror ("Unable to fold! Inconsistent input.");
   return R;
};

function body getfirstinterval(P: P_element): Interval
{   if (empty (P)) {
      perror ("Unable to get the first interval of periodic element!");
      return interval (zero (), zero ());
   };
   if (elementempty (P.aperiodicpart))
      return intervalfirst (P.periodicpart);
   else
      return intervalfirst (P.aperiodicpart);
};
```

```
function body getinterval(P: P_element,
                          i: integer): Interval
{   o2 Interval I;
    int j;

    if (i <= 0) {
        perror ("Unable to get interval!");
        return I;
    };
    if (intervalcount (P) < i) {
        perror ("Unable get interval from periodic element!");
        return I;
    };
    if (count (P.aperiodicpart) >= i)
        return intervalget (P.aperiodicpart, i);
    else {
        j = (i - count (P.aperiodicpart)) % count (P.periodicpart);
        j = ((j==0) ? count (P.periodicpart) : j);
        return intervaltranslate (intervalget (P.periodicpart, j));
    };
};

function body getlastinterval(P: P_element): Interval
{   if (empty (P)) {
        perror ("Unable to get the last interval of periodic element!");
        return interval (zero (), zero ());
    };
    if (elementempty (P.periodicpart))
        return intervallast (P.aperiodicpart);
    else
        return interval (infinity(), infinity());
};
```

9.6.2 Temporal Transformations

```
function body scale(P: P_element,
                    d: Duration): P_element
{   o2 P_element Q;
    o2 Instant    t;

    if (d == 0) {
        perror ("Unable to scale by a factor of 0!");
        return Q;
    }
```

```
   if (empty (P))
      return P;
   Q.aperiodicpart = elementscale (P.aperiodicpart, d);
   if (elementempty (P.aperiodicpart))
      t = 0;
   else
      t = elementbegin (P.periodicpart) - elementbegin (P.aperiodicpart);
   Q.periodicpart  = elementtranslate (elementscale (P.periodicpart,  d),
                  t * d - t);
   Q.period       = d * P.period;
   return Q;
};

function body translate(P: P_element,
                   d: Duration): P_element
{   return tuple
         (   aperiodicpart: elementtranslate (P.aperiodicpart, d),
             periodicpart : elementtranslate (P.periodicpart, d),
             period        : P.period);
};
```

9.6.3 Temporal Comparisons

```
function body after(P: P_element,
                Q: P_element): boolean
{   return (instantafter (begin (P), end (Q))); };

function body before(P: P_element,
                 Q: P_element): boolean
{   return (instantbefore (end (P), begin (Q))); };

function body contain(P: P_element,
                  Q: P_element): boolean
{   o2 tuple (a:P_element, b:P_element) R =  synhronize (P, Q);

   return (elementcontain (R.a.aperiodicpart, R.b.aperiodicpart) &&
         elementcontain (R.a.periodicpart,  R.b.periodicpart));
};

function body meet(P: P_element,
               Q: P_element): boolean
{ return (instantequal (end (P), begin (Q))); };

function body metby(P: P_element,
```

```
                 Q: P_element): boolean
{ return (instantequal (end (Q), begin (P))); };

function body overlap(P: P_element,
                  Q: P_element): boolean
{   o2 tuple (a:P_element, b:P_element) R =  synhronize (P, Q);

   return (elementoverlap (R.a.aperiodicpart, R.b.aperiodicpart) ||
          elementoverlap (R.a.periodicpart,  R.b.periodicpart));
};

function body overlappedby(P: P_element,
                      Q: P_element): boolean
{ return overlap (Q, P); };

function body containedby(P: P_element,
                     Q: P_element): boolean
{   return (contain (Q, P)); };
```

9.6.4 Set Theoretic Operators

```
function body Intersect(P: P_element,
                   Q: P_element): P_element
{   o2 P_element R;
   o2 tuple (a:P_element, b:P_element) aux;

   if (empty (P) || empty (Q)) {
      return R;
   };
   aux = synhronize (P, Q);
   R.aperiodicpart = elementintersect (aux.a.aperiodicpart, aux.b.aperiodicpart);
   R.periodicpart  = elementintersect (aux.a.periodicpart,  aux.b.periodicpart);
   R.period        = aux.a.period == 0 ? aux.b.period : aux.a.period;
   return normalize (R);
};

function body Union(P: P_element,
                Q: P_element): P_element
{   o2 P_element R;
   o2 tuple (a:P_element, b:P_element) aux;

   if (empty (P))
      return Q;
   if (empty (Q))
```

```
      return P;
   aux = synhronize (P, Q);
   R.aperiodicpart = elementunion (aux.a.aperiodicpart, aux.b.aperiodicpart);
   R.periodicpart  = elementunion (aux.a.periodicpart,  aux.b.periodicpart);
   R.period        = aux.a.period == 0 ? aux.b.period : aux.a.period;
   return normalize (R);
};

function body complement(P: P_element): P_element
{   o2 P_element R, Q = reduce (P);
   o2 Instant t = zero ();
   o2 Interval I;
   o2 integer  i;
   /* assuming no adjacent intervals */

   if (empty (Q))
      return Time ();
   for (I in Q.aperiodicpart) {
      if (! instantequal (I.begin, t))
         R.aperiodicpart += list (interval (t, I.begin));
      t = I.end;
   };
   R.period = Q.period;
   if (elementempty (Q.periodicpart)) {
      R.periodicpart = list (interval (t, instanttranslate (t, 1)));
      R.period       = 1;
   } else {
      if (! instantequal (Q.periodicpart[0].begin, t))
         R.aperiodicpart += list (interval (t, Q.periodicpart[0].begin));
      t = Q.periodicpart[0].end;
      for (i=0; i<count(Q.periodicpart)-1; i++)
         if (! instantequal (Q.periodicpart[i].end, Q.periodicpart[i+1].begin))
            R.periodicpart += list(interval(Q.periodicpart[i].end,
                                 Q.periodicpart[i+1].begin));
      if (! instantequal(intervallast (Q.periodicpart).end,
                   instanttranslate(elementbegin(Q.periodicpart), Q.period)))
         R.periodicpart += list (interval (intervallast (Q.periodicpart).end,
                   instanttranslate(elementbegin(Q.periodicpart), Q.period)));
   };
   return R;
};

function body difference(P: P_element,
                     Q: P_element): P_element
```

```
{    return intersect (P, complement (Q)); }
```