An Object-Oriented Data Model for a Time Series Management System

Werner Dreyer, Angelika Kotz Dittrich, Duri Schmidt
{dreyer, dittrich, schmidt}@ubilab.ubs.ch

UBILAB, Union Bank of Switzerland, Zurich

Abstract

The analysis of time series is a central issue in economic research and many other scientific applications. However, the data management functionality for this field is not provided by general-purpose DBMSs. Therefore, we propose a data model of a specialized Time Series Management System (TSMS) which accounts for these needs. The model is centered around an object-oriented kernel that offers the classes and value types needed for the target applications. The model provides base classes for multivariate time series and for groups as a means to hierarchically partition the time series space. The system offers a computationally complete data manipulation language including capabilities to query time series and groups. An elaborate array model is supported to account for the functional needs of statistical computations. Furthermore, a customizable calendar system providing a variety of predefined calendars is included.

1 Introduction

Time series management is important for many research areas such as economic and finance research, but also for various non-research related activities like portfolio management in banking. The reason for this importance is the increasing application of empirical methods with more and more data. However, the data intensity causes several problems (see [4] for a detailed discussion):

- Time series bases contain thousands of time series. For this data volume, ad hoc management is inadequate. Traditional database management systems do not provide the appropriate functionality for time series management, either.
- Statistics programs provide sophisticated analysis methods. Yet, they are not designed for the management of large quantities of time series data.
- Existing systems do not provide reasonable search mechanisms for finding relevant time series.
- Many tools in which time series are used do not provide the functionality necessary for filtering and transformation existing time series, nor for computing new time series out of existing ones.

According to our opinion, the commercially available time series management systems do not address all the relevant issues adequately. Furthermore, earlier research work in related fields, namely in temporal and statistical databases, does not sufficiently consider the special requirements of time series.

Considering the problems and the available solutions, we decided to develop a new, object-oriented time series management system. The following paper presents the data model of this system. Firstly, we discuss an example of a time series database. Next, we give a survey of the main features of the data model. Then, we discuss related work. Afterwards, we explain the data model in detail. Finally, we present our conclusions and future work.

2 Example: the Zurich Stock Exchange time series base

In this section, we describe a time series base containing daily securities prices of the Zurich Stock Exchange. We will use this example throughout the article to illustrate various concepts.

2.1 Characteristics of a time series

In our example, financial researchers are interested in collecting the following data:

- The name, security number, start date, end date, calendar, and total trading volume. These data are valid for the time series as a whole.
- The daily opening, closing, high, and low prices, plus the daily trading volume. These data are valid for a single day.

Fig. 1 shows an example of such a time series.

2.2 Categorization

A time series base which models the entire Zurich Stock Exchange would contain thousands of different securities, and therefore thousands of time series. This makes it difficult for a database user to find the time series he or she needs for a particular task. For our purposes, we need to categorize these time series (group them according to certain characteristics). An obvious categorization

This article appears in the Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management; Charlottesville, Virginia USA, Sept. 28-30, 1994.

Name:	UBS registered
Security_number:	136 102
Start_date:	20.12.93
End_date:	23.12.93
Calendar:	Business week
Total_trading_vol.:	90869

Date	Open	Close	High	Low	Daily_vol.
20.12.93	319	323	324	319	23249
21.12.93	322	328	329	322	19403
22.12.93	328	330	331	327	35845
23.12.93	331	328	331	328	12372

Fig. 1: An example time series

would be to apply the same categories as in the stock exchange price lists of newspapers, for example:

- Swiss vs. foreign securities
- Type of security: stocks, bonds, options, indices, …
- Industry: banking, insurance, transport, chemistry, …

Fig. 2 shows part of such a categorization structure. The names written in italics are time series, the other names are categorization criteria. Fig. 3 shows the "Banking"-group in more detail.

This example shows only satirically and hierarchically grouped time series. A financial analyst could also group the time series according to his or her individual criteria like the performance of the shares (such as above, equal to, or below an index). This categorization should be usable in coexistence with the previously described structure, so that individual time series can belong to different categories. Besides, categorization according to performance is dynamic because this structure may vary daily.

3 Main features

In the following, we will give a survey of our data model. We will later discuss the characteristics in detail and present examples where appropriate. The main features are as follows:

- **Specialized object-oriented data model with time series and group as root classes:** Our data model is object-oriented and it is specialized for the domain of time series management. It has two base classes: the time series class and the group class. These classes have a rich functionality adapted to the problem domain. Therefore, a user is not concerned with the implementation of the basics of time series or groups but only with their adaptation to his or her special needs.
- **Objects and values:** Our data model is a hybrid data model with classes[1] and value types. The distinction of objects and values allows object and functional programming style to be used where they are more appropriate. Furthermore, the overhead of objects can be avoided where object semantics is not necessary.
- **Multivariate time series with query capabilities and time scale conversion as basic abstraction:** Multivariate time series are pivotal for the problem domain addressed by this project. For this reason, the basic time series class is designed for the modeling of multivariate time series. Furthermore, queries can be executed on time series and time scale conversion can easily be done. This built-in functionality solves common time series problems without requiring the user to make any further implementation.
- **Groups as an effective categorization and aggregation instrument:** A large time series base may contain thousands of time series. Without partitioning all these time series according to different criteria, it would be difficult for users to find the time series relevant to their work. In our data model, groups serve the purpose of partitioning. They have a member set consisting of arbitrary time series and groups. Queries may be executed on the member set and set operations are provided to manipulate it. Groups do not only function as a means of categorizing. Because of their methods, groups can also be used as a flexible means to do computations on their members, such as aggregate some value over all the members.
- **Important role of arrays:** Matrix algebra plays a central role in statistics. In our data model, this central role is reflected by the functionality of arrays. Arrays can dynamically change their number of dimensions and the size of the dimensions if not defined otherwise. A rich set of operations, such as arithmetic, relational or logical operations, is defined for whole arrays. Finally, time series, groups or records can be accessed as arrays where appropriate. This rich array functionality makes it easy to implement statistical methods for transformation and filtering.
- **Simple records:** Records are important building blocks for time series and groups. Records may contain as elements simple values like integers or arrays of simple values. Record elements can be accessed in the traditional way via record labels. In order to be able to implement generic functions which work on different record types, access by index and access as arrays are also provided. This simple record design is easy to understand and provides adequate modeling power for our problem domain.
- **Extensive calendar support:** A calendar is attached to every time series. It maps time points to the corre-

1 We use the term class to denote the type of an object as opposed to the type of a value. The word class does not imply - as is sometimes assumed in database terminology - the extension of all instances of the type.

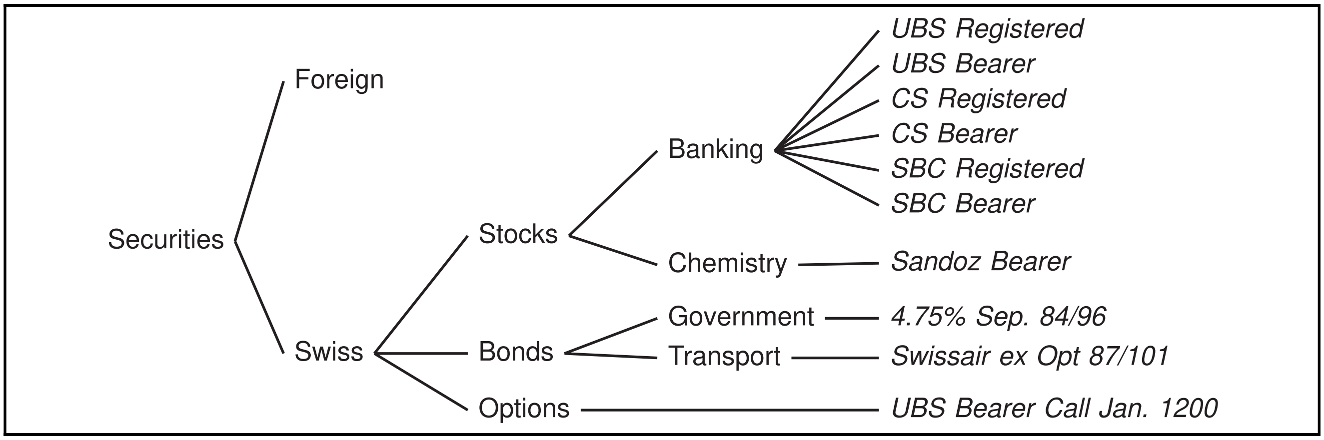

Fig. 2: A structured securities time series base

sponding data. Our system knows about various calendars, such as Gregorian calendar, national calendars, business week calendar, etc. The calendars can be customized and sophisticated date arithmetic is supported. The extensive calendar support is crucial for a TSMS because it is the base for the often necessary time scale conversion and for temporal queries.

- **Sound data manipulation language suited for occasional and experienced users:** The data manipulation language provides adequate features for the implementation of functions and methods. Specialized control structures facilitate the manipulation of time series and groups. In order to further accommodate occasional users, our data manipulation language uses adaptive type checking, a combination of static and dynamic type checking. Together with the flexibility and the rich built-in functionality of our data types, also occasional users with little programming knowledge can solve demanding time series problems.

Name:	Banking
No_of_members:	6
Members:	UBS_Registered
	UBS_Bearer
	CS_Registered
	CS_Bearer
	SBC_Registered
	SBC_Bearer

Fig. 3: An example group

4 State of the art

The data model of a TSMS is located between temporal data models on the one side and models for statistical and scientific databases on the other.

Most models developed in the field of *temporal databases* (for an extensive overview see [15]) do not specifically address the problems of time series. The most obvious difference is that these models use an interval approach (useful for answering questions like "When were certain facts true in the past?") while time series management asks for the recording of historical events associated with discrete points in time and applying statistical methods to these events. *Statistical databases* and *scientific databases* [11], [7] address certain problems that a TSMS has to deal with, too. However, they give no preference to the time dimension and do not offer special functionality for the handling of time series like time scale conversion etc.

To our knowledge, only one model has been published so far that comes rather close to ours, namely the model by Segev et al. in [12] and [13]. As might be expected from the application requirements, their notion of time series is similar to our model as far as the basic features are concerned. The main differences are as follows: Segev et al. develop a time series database along the lines of an extended relational DBMS. Being based on a predefined set of operations that may be applied in a declarative way, their model offers only restricted expressional power. In contrast, our model is based on the object-oriented approach where the stress lies on user-defined functionality and computational completeness [1]. Arbitrary kinds of time series and groups can be modeled as classes profiting from features like inheritance, reusability and extensibility. The complete spectrum of time series transformation, filtering, statistical evaluation etc. can be handled uniformly. For example, the model in [12] does provide an interpolation function for time series (the "type" of a time series), but neither an aggregation function nor individual interpolation functions for different attributes of the same series. Extensions like these are easily feasible in our model.

Another major difference is that the notion of "Concept" found in [12] is not equivalent to what is called a group in our model. Concepts try to combine three ideas: inheritance (IS-A hierarchy of Concepts), the grouping of time series according to some common feature or usage and a view mechanism based on event construction. In our model, these features are kept separately and may be combined as needed (inheritance is supported by the class hierarchy, groups serve to collect time series into an arbitrary set hierarchy, event construction is done by applying transformation methods). Furthermore, events within one time series can be combined more flexibly as we do not require all attributes of an event to have the same interpolation function.

Beside looking into related research work, we also evaluated solutions for time series management currently applied in practice. Solutions based on file systems or relational databases are common, but in both cases the data management functionality is unsatisfactory, a major part of it being implemented inside the applications. There are very few commercial DBMSs specialized for time series management (e.g. the FAME system [6]). These products offer some really useful special features, but do not offer a comprehensive solution including powerful data modeling, general retrieval facilities, mechanisms for data quality management etc. From a practical point of view, there is an obvious need for more research in the field of time series data management.

5 Type system of the data model

This chapter will describe the type system of the TSMS data model in detail. As has been motivated in chapter 3, the model is hybrid in that it supports value types and classes. We start by describing the value types being the basic constituents and continue with the presentation of the classes for time series and group objects.

5.1 Value types

General characteristics: Value types serve as elementary building blocks from which classes are constructed. The header type and event type of a time series class, for example, are record types. Accordingly, a time series consists of a record value as the header and a sequence of records as the events. Similarly, a group consists of a record value as the header and a set of references.

In contrast to objects as instances of classes, values as instances of value types do not have identity. Consequently, it is not possible to share values. In value assignment or in passing values as parameter therefore always copy semantics are applied. Values do not have methods. However, a rich set of operators and functions is provided for their manipulation and more functions may be implemented by the users.

The reasons for the distinction between classes and value types are as follows: It is a natural differentiation for the types of our problem domain. According to our valuation, object-based programming is a very adequate programming style for the treatment of time series and groups. However, because values (and especially arrays) play a central role as elements of algebraic computations, it is sensible to make them suitable for a more functional programming style, i.e. to define basic operations such as arithmetic on whole arrays.

Our data model contains three value type families: simple value types, array value types and record value types. The latter two type families are types which are composed from simple value types.

Simple value types: The simple value types family currently includes the data types character, integer, float, time stamp, time span, and reference. For all these data types, the usual operations are predefined, i.e. for integer and float, there are arithmetic, relational and logical operations, for character, there is comparison, for time stamp and time span, there is date arithmetic, and for reference, there are the equality and the dereference operation.

Reference types are further divided into reference to time series, reference to group, and unspecified reference. An unspecified reference can refer either to a time series or a group. References to time series and groups can be more specialized. They can be defined to refer to an instance of a specific class. A reference variable defined for a certain class may also be assigned a reference to an instance of a direct or indirect subclass.

Array types: Arrays play a central role in our data model because matrix algebra is of pivotal importance for statistical methods like filtering. Our array model is influenced by the APL-array model [8] [9]. For every simple type, a corresponding array type is defined. The type of an array is set at definition time and cannot be changed. The number of dimensions and the size of each dimension, i.e. the shape of an array, are instance properties and they are either restricted at definition time or they are variable. If a user specifies restrictions concerning the shape more checking at compile time can be done.

The same operations as for simple types are also defined for the corresponding array types For arrays of integers, for example, there are arithmetic, relational and logical operations. Therefore, a user can formulate matrix algebra operations on a high, convenient level of abstraction without being concerned with how to iterate over all the elements (see example).

Other operations are defined to manipulate the structure of an array. Examples are functions to reshape an array, to ravel the elements of an array or to transpose an array. Furthermore, a variety of selection operations provide access to the elements (see example).

```
// definition of 4 arrays with 3 rows and 4 columns
int A[3 4]; int B[3 4]; int C[3 4]; int D[3 4];
```

```
// Arithmetic with whole, compatible arrays
A = B + C - D;
// select the elements of the intersection of rows 1 3
// and columns 1 4 2
A[1 3; 1 4 2]
```

Record types: Record types are used for the definition of the header of time series and group classes as well as for the definition of the event type of time series classes. Our record model is simple. The elements of a record type may be either simple value types or array types. Therefore, record types are flat, i.e. record types cannot be elements of record types. Our requirements analysis showed that this record model provides adequate modeling power for the problem domain and that the additional modeling power of a nested record model is not required.

Because records are intended to be mostly used in conjunction with time series classes and group classes, new record types are only implicitly defined via the definition of time series or group headers and via the definition of the event structure.

Two record types are equal and their instances are assignment compatible if they have the same number of elements and the corresponding elements have compatible types.

We provide assignment, i.e. element wise copy, and test of equality as operations for records. Other operations may be carried out on records by accessing their elements.

The most simple way of accessing record elements is by their label (see example).

In order to make it possible to write generic functions working with different record types, all elements also have an index and the elements may be accessed by this index (see example).

Finally, because of the importance of arrays, records may be accessed as arrays if the accessed elements have homogeneous data types, and in the case of arrays, if they have the same shape (see example). With these various access variants, records can be used in a flexible way. In other languages, records merely coexist with other types. In contrast, our access features lead to a tight integration of the two types.

```
// let "aRecord" be a record with the elements
// "high", "low"
// access of the record element "high" by label
aRecord.high
// record element access by index
aRecord.<1>;
// access of the record elements "high" and "low" as
array; returns a vector of 2 elements
aRecord.[high low];
// access the whole record as an array; returns a
// vector of 2 elements
aRecord.[]
```

5.2 Classes

General characteristics: Our data model is a specifically tailored object-oriented model that supports the central concepts time series and group. On the one hand, it provides all the benefits of an object-oriented model like inheritance and polymorphism. Modeling by classes and objects is intuitive and easy to understand and the functionality can be extended and adapted to user needs. On the other hand, the user does not have to cope with a general object model, but with specific abstractions for time series management. The common functionality of the problem domain like calendar functionality, data aggregation, categorization etc. is predefined, such that the user may concentrate on application-specific details.

Technically speaking, there are two root classes: Timeseries and Group. From these, subclasses with more specialized functionality for time series and groups can be derived recursively. Each time series or group is an instance of a class derived from the root classes. Class derivation is by single inheritance because ease of use is our foremost aim and the targeted applications do not exhibit a strong need for multiple inheritance.

Each time series and group is an object with a unique identity by which it may be referenced and shared. Each class has a name, a superclass, a header description and a number of methods as well as further elements that are different for time series and groups.

In the sequel, a general formalization of the class structures is presented. We do not introduce a concrete DDL (data definition language) syntax as the TSMS will support data definition via direct manipulation with graphical editors.

Time series classes: Time series classes serve to model multivariate time series. The corresponding root class comes with all the functionality for the manipulation, retrieval, and conversion of such time series. By using the various access features, it is easy to apply statistical analysis.

A *time series class* is defined to be a 5-tuple:

```
(name, superclass name, header type,
event type, {methods}, calendar)
```

name is a string denoting the name of the time series class.

superclass name is the name of the time series class from which this class is derived. For the root time series class, this is not defined. A derived time series class inherits from its superclass the header type, the event type, the calendar as well as the set of methods. Additionally, the header type and the event type may be enhanced by further attributes. Inherited methods may be overwritten and additional methods may be defined. Overloading of methods with different parameter types is also supported. The calendar may not be overwritten, i.e. as soon as a calendar is defined for a class, all direct and indirect subclasses of this class have the same calendar. The reason is that the implementation of methods that are not overwritten may rely on the calendar.

header type is a record type defining the name and data type of the attributes describing each time series as a whole. In the root class, there are some predefined attributes which are the same for all time series classes (e.g. an attribute name). Derived classes may have arbitrary further attributes.

event type is a record type defining the name and type of each attribute in the multivariate time series events.

{methods} is a set of methods, each defined by specifying method name, formal parameters, return type and implementation (i.e. associated code). There are a number of predefined methods in the root time series class.

calendar is the calendar associated with time series of this type. The calendar serves to interpret the time points of the events. The calendar of a class may be undefined. However, in this case, the class can only be used to derive further subclasses and may not have instances of its own.

A *time series* is a 4-tuple of the form

```
(time series class name, header, events,
start time stamp)
```

time series class name identifies the class of the time series (as described above).

header is a record of the attributes that characterize the whole time series.

events is a sequence of records, each record representing one event.

start time stamp is the time point of the first event. The time stamps of all following events are interpreted with respect to this time stamp and the calendar of the time series class.

For an example of a time series see fig. 1 in chapter 2.1. The upper part of the picture shows the header, the lower part the sequence of events. The header attributes Start_date and Calendar are implicit in the model.

Direct time series data access: The elements of a time series may be accessed directly in various ways. All access constructs presented below can be used on the left hand side as well as on the right hand side of an assignment. In the sequel, we distinguish between three kinds of access to a time series: access to the header, access to single event records and access to multiple events by conversion to arrays.

Header access: The header of a time series can be accessed just like a record using a dot notation, for example:

```
// returns the whole header of time series
// Sandoz_Bearer
Sandoz_Bearer.
// returns the header attribute Start_date
Sandoz_Bearer.Start_date
```

Single event access: Single events can be accessed as records quite similar to the header access. The event is indexed by specifying an expression returning an index or a time stamp, for example:

```
// returns the 100th event of time series
// UBS_Registered
UBS_Registered [100]
// returns the low price of UBS_Registered
// on the 1/1/93
UBS_Registered [1/1/93].Low
```

As in any record (cf. chapter 5.1.4), homogeneous attributes of an event may be accessed as an array. The following example selects the event of January 1st, 1990, from time series CS_Bearer and returns its Open and Close price as a 2-element array:

```
CS_Bearer [1/1/1990].[Open Close]
```

While access via record elements will mainly be used for element wise modification and assignment, the conversion into arrays will be used for computations and statistical analysis.

Multiple event access: It is also possible to access a sequence of several events (or of selected homogeneous attributes thereof) as an array. For this kind of access, the time series is indexed by a one dimensional array or by an interval of integer or time stamp values. The resulting array is obtained by first converting the single events into arrays (see above) and then concatenating them along a further dimension.

```
// Return a 4 x 2 matrix with the values of Open
// and Close in the 1st, 3rd, 5th and 10th event
CS_Bearer [1 3 5 10].[Open Close]
// Return an array with all events between
// January and December 1990
CS_Bearer [1/1990 .. 12/1990 ].[]
```

Predefined methods: Common operations on time series are predefined as methods of the root class Timeseries. These methods provide the basic protocol to update and query time series. This protocol may be changed and/or enhanced in derived classes by overwriting and adding methods as needed. In the sequel, we will present the most important parts of the functionality, including methods for modifying time series, retrieval of events, time scale conversion as well as meta data access.

Modification of time series: There are methods for adding, removing and updating the events recorded in a time series. Addition of new events as well as removal of events may happen either at the beginning or at the end of the time series. Any event within the time series may be modified. In addition, a whole time series or part of it may be copied to another time series (provided both time series have compatible calendars).

Examples:[2]

```
// For the example, we first define and initialize
// a record ev of the event type of class Security
// (for the variable declaration see chapter 6.1).
EventTypeOf (Security) ev = <319, 323, 324,
319, 23249>;
// Insert one new event (i.e. a compatible record) at
// the end/the beginning of
// time series UBS_Registered.
UBS_Registered→Append (ev)
UBS_Registered→Prepend (ev)
// Remove all events that have been recorded
// forJanuary 1993 or later.
UBS_Registered→RemoveFrom (1/1993)
// Remove the first 20 events.
UBS_Registered→RemoveFirst (20)
// Modify the 100th event of UBS_Registered. The
// new attribute values of this event will be those of
// the variable ev defined above.
UBS_Registered→Update (100, ev)
// Copy 200 events from UBS_Registered to UBS1
// starting on 1/1/1990.
UBS_Registered→Copy (UBS1, 1/1/1990,
200)
```

Event retrieval: For queries on time series, i.e. selection of events satisfying a condition, a set of retrieval methods is provided. There are methods that select a vector of the indices or time stamps of all events satisfying a given condition (selectIndex and selectTimestamp methods). You may also choose to retrieve only the first event fulfilling a condition (detectIndex and detectTimestamp methods).

As an example, think of selecting all events from time series SBC_Registered where Low is smaller than 250. As the result, an array of time stamps is returned.

```
SBC_Registered→SelectTimestamp ("Low <
250")
```

The following expression is used to find the index of the first event after 1st January, 1990, where opening and closing price were the same. The expression will return the index of the event as an integer value or 0 if no event was found). The attribute $timestamp used in the condition is implicit in every event.

```
SBC_Registered→DetectIndex ("$timestamp
> 1/1/1990 and Open == Close")
```

Time scale conversion: A transformation method is provided to convert a time series into another one, taking into account different time scales, i.e. calendars with different granularity. Conversion involves either aggregation (from finer to coarser granularity, like daily to monthly data) or interpolation (from coarser to finer granularity, like quarterly to monthly data).

In order to support time scale conversion to coarser granularity, an aggregation property similar to the "observed property" in [6] can be defined for each event attribute. This property can be set as undefined, begin or end of period stock value, flow value, average value, maximum or minimum value. The default setting is undefined. Depending on this property, the system will automatically calculate the aggregated values for a coarser grained time series unless another aggregation function is explicitly specified with the conversion method.

For the conversion to a shorter periodicity one would also like such an automatism. Unfortunately, requirements analysis shows that there is no such simple relationship as for aggregation, because interpolation depends on the individual data rather than the type of time series. For example, for stock values, the interpolation function to be applied depends on the actual growth of the series which may have been linear, exponential or other. For this reason, the model does not provide a default interpolation property for event attributes, but the interpolation functions is always required as a parameter of the convert function.

In the following example, we show how the daily time series UBS_Bearer can be aggregated into UBS_Bearer_monthly. The aggregation properties defined on UBS_Bearer are

```
Low: minimum
High: maximum
Open: beginOfPeriod
Close: endOfPeriod
```

The following statement will aggregate events automatically, calculating the monthly minimum for Low, the monthly maximum for High, the first value in month for Open and the last value for Close:

```
UBS_Bearer→Convert (UBS_Bearer_monthly)
```

The next example shows the reverse conversion. Linear interpolation is applied for attribute Low, square interpolation for attribute Close.

```
UBS_Bearer_monthly→Convert (UBS_Bearer,
        " linear Low, . . ., square Close")
```

Group classes: A further important abstraction of the model is the group concept. By the use of groups, the large number of time series usually present in a time series base can be partitioned. Partitioning can be done hierarchically as groups may contain other groups recursively. A time series may be part of several groups according to different partitioning criteria. Group classes offer all the functionality for manipulating, querying and applying set operations to groups.

A *group class* is defined as a 3-tuple:

```
(name, superclass name, header type,
{methods})
```

2 The syntax for method invocation is
receiver object→methodName (parameterList)

name, superclass name, header type and {methods} are defined analogous to the time series classes above. Group classes do not have any further elements because the member sets of all groups are of the same type (a set of references) and need not be declared explicitly in the class.

A derived group class inherits from its superclass the header type and the set of methods. On class derivation, the header type may be enhanced by further attributes, inherited methods may be overwritten and additional methods may be defined.

A *group* is defined as a 3-tuple of the form

```
(group class name, header, member set)
```

group class name identifies the class of the group.

header is a record of the attributes that characterize the whole group.

member set is a set of references to time series and groups. By that, heterogeneous groups can be built and groups can be nested to arbitrary depth.

For an example of a group, see fig. 3 in chapter 2.1. The upper part of the picture shows the header, the lower part the member set.

Predefined methods and functions: For groups, direct access to header attributes is the same as for time series, like in the following example:

```
// returns the whole header of group Swiss_Bonds
Swiss_Bonds.
```

The basic protocol for update, retrieval and set operations on groups is predefined by the root class Group. As for time series, the protocol may be changed and/or enhanced in derived classes by overwriting and adding methods as needed. We will give an overview of the functionality, regarding modification of groups, retrieval of members and set-theoretic operations. Like time series, groups also have methods for meta data access which we will not deal with in detail.

Modification of groups: Methods are provided to manipulate the member set of a group, i.e. to add and remove elements, for example:

```
// Make Sandoz_Bearer a member of
// Swiss_Chemistry_Stocks.
Swiss_Chemistry_Stocks→Add
(Sandoz_Bearer)
// Make all direct members of Ciba_stocks
// members of Swiss_Chemistry_Stocks.
Swiss_Chemistry_Stocks→AddMembers
(Ciba_Stocks)
// Make all time series which are direct or indirect
// members of Swiss_Chemistry_Stocks members
// of Swiss_Stocks.
Swiss_Stocks→AddAllTS
(Swiss_Chemistry_Stocks)
```

```
// Remove Sandoz_Registered from
// Swiss_Chemistry_Stocks.
Swiss_Chemistry_Stocks→Remove
(Sandoz_Registered)
```

Note that there are group methods operating on one element (like Add), methods with "flat" semantics on the direct members (like AddMembers) and methods with "deep" semantics dealing with all time series recursively (AddAllTS). In the latter case, if the same member is encountered several times, it is dealt with only once. Cyclic references are cut off by this rule.

Retrieval of group members: The member set of a group can be queried using the Select and Detect methods. The Select methods return a vector of references to members fulfilling a given condition. The Detect methods just return a reference to the first member found.

The following expression retrieves the references to all time series in group Swiss_Government_Bonds that started in May 1993. The result of the method invocation is an array of references. The second parameter of this method is optional. It specifies the sort order of the result, in this case sorting is by start date in descending order.

```
Swiss_Government_Bonds→SelectTS
        ("Start_date ≥ 1/5/1993 and
         Start_date ≤ 31/5/1993",
         "Start_date descending")
```

The next example shows the retrieval of a group member, i.e. either a time series or a subgroup, with name 'UBS Registered'. Only one such member is to be detected.

```
Swiss_Banking_Stocks→Detect (" name =
        'UBS Registered' ")
```

Set operations: Set operations are central in the handling of groups. Therefore, functions are provided to calculate the union, intersection and difference of groups. Analogous to the select operations, the result of set operations is delivered as an array of references. To apply set functions recursively, their parameters may be either groups or arrays of references.

```
// Return the union of Swiss banking and chemistry
// stocks.
union (Swiss_Banking_Stocks,
Swiss_Chemistry_Stocks)
// Return the Swiss banking stocks which are not
// above index.
difference (Swiss_Banking_Stocks,
Above_index)
```

6 Further characteristics of the data model

6.1 Calendar functionality

For time series management, elaborate calendar functionality is obviously a central issue. The calendar functionality has to take into account a variety of ways to impose structure on time. Flexibility must be provided for application-specific needs. In our system, we take an approach similar to [14] in that a number of various calendars are defined within a general framework. However, our approach is not tied to a specific language like SQL and in that more resembles [3]. Taking into account the special needs of time series management, we define calendars which differ according to the criteria:

- base calendars
- granularity
- business days
- national and regional holidays

There are various *base calendars* like the Julian, Gregorian or Islamic calendar. In addition to these natural calendars, there are artificial calendars, e.g. a regular calendar with 360 days per year and 30 days per month, an ordinal calendar just counting time units as a sequence of natural numbers and an enumerated calendar which consists of an irregular sequence of single time points specified by the user.

According to various application areas, calendars vary in their *granularity*, i.e. the chronon they support (for terminology cf. [10]). Chronons currently supported in calendars are days, weeks, months, years and multiples thereof.

Calendars may have the *mode "business" or "non-business"*. A business mode calendar does not record all days of a base calendar but working days only.

National and regional calendars take into account different local holidays. Such calendars can easily be obtained by parameterizing other calendars with a set of holidays.

Example:

```
Calendar
        → base: Gregorian
        → time unit: days
        → mode: business
        → holidays: Zurich regional
```

The TSMS calendar model allows to

- *define all these kinds of calendars*: Currently, the data model offers a number of predefined calendars which may be parameterized with specific holidays. Arbitrary user-defined calendars are not yet supported by the model, though the underlying calendar system has already been designed for this kind of extension.
- *transform time units from one calendar to another*: Time stamps and time spans can be transformed between calendars within the limits of calendar compatibility. For example, a time stamp of the daily Gregorian calendar may be transformed to the monthly Gregorian calendar (yielding the month and year of the date) and vice versa (yielding an array of all days in the month).
- *apply relational operators to time*: Time stamps can be compared with respect to their occurrence in time (before, after, at the same time). Time spans can be compared as to their duration (longer, shorter, same duration).
- *scan calendars*: It is possible to iterate over all or part of the time stamps in a calendar. For example, depending on the kind of calendar, one may iterate over all months, all working days, all days after/before a given date etc. The data manipulation language offers increment and decrement operators to use this functionality.
- *do arithmetic operations on time*: Frequently needed arithmetic operations are supported for time stamps and time spans. For example, a span may be added or subtracted from a time stamp, the span between two time stamps may be calculated etc. The arithmetic operations take into account holidays, weekends, length of months, leap years etc.

Each time series is associated with a calendar that serves to map its events to the corresponding points in time. For each time series, a start time stamp is specified (see chapter 5.2.2). The sequence of events starts with this time stamp and is implicitly incremented by the time granule of the calendar (for enumerated calendars the time stamps are explicitly recorded).

Example:

```
calendar = <Gregorian / day / business / Zu-
rich regional >
start = 1/1/1993
// The 1st and 2nd Jan. are holidays in Zurich,
// the 3rd Jan. 1993 was a Sunday)
event 1        =>        4/1/1993
event 2        =>        5/1/993
event 3        =>        6/1/1993
event 4        =>        7/1/993
event 5        =>        8/1/993
event 6        =>        11/1/1993
. . .
// Calendar calculation takes into account business
// days and holidays, for example:
 4/1/1993 + 10 time units => 18th Jan. 1993
```

6.2 Language aspects

Methods and global functions: The data model offers a computationally complete manipulation language that serves to define methods on time series and group classes as well as global functions.

Adaptive type checking: The language supports adaptive type checking, i.e. a combination of static and dynamic checking, for records, arrays and references. The idea is to provide static type checking wherever possible without hindering flexibility where needed. Dynamic type checking is feasible because type information can be obtained from the database at runtime.

Control structures: The language offers the common control statements (if, switch, while, return, for, etc.). Besides, it supports a number of special loop statements to iterate over all events of a time series or the whole member set of a group.

6.3 Time series bases

Time series bases are a means to coarsely partition the universe of time series. They are the analogue of data bases in a general DBMS. Each time series or group belongs to exactly one time series base. In each time series base, there is a system-defined group serving as a root from which all time series and groups of this base may be reached.

Time series bases serve as unit of visibility and access control. Read and write access for individual users and user groups is granted on the basis of time series bases. In a typical environment, there will be public and private time series bases.

7 Conclusion and future work

In this paper, we presented a specialized object-oriented data model for time series management. We showed that the model offers multivariate time series and recursive groups as the main class abstractions as well as a variety of value types with arrays playing the most important role. The model features elaborate calendar functionality, aggregation properties for time scale conversion and a powerful data manipulation language to define methods and global functions.

We are currently implementing this data model in a time series management system based on an off-the-shelf object-oriented DBMS and the application framework ET++ [16]. A kernel version of the TSMS including the root classes, the value types as well as the calendar system is operable. The next implementation steps will be targeted at the graphical user interface, enhanced functionality for array types and an interpreter for the data manipulation language.

Future work will relate to

- interoperability issues and data exchange in an environment with heterogeneous data sources and client applications [5]
- data quality management
- the introduction of physical properties for value types and classes as a means for efficient data storage and retrieval (e.g. physical event order, linearization order of arrays, compression techniques to be applied, expected size of time series, etc.)
- language support and tools for user-defined calendars.

Of course, a central task will also be to come up with results from the practical experiences of applying the TSMS in various banking applications.

References

[1] R.G.G. Cattell: Object Data Management - Object-Oriented and Extended Relational Database Systems. Addison-Wesley, 1991.

[2] R. Chandra, A. Segev: Managing Temporal Financial Data in an Extensible Database. Proc. of the 19th VLDB Conf., Dublin 1993.

[3] R. Chandra, A. Segev, M. Stonebraker: Implementing Calendars and Temporal Rules in Next Generation Databases. Proc. of the Intl. Conference on Data Engineering, Houston, Texas, February 1994.

[4] W. Dreyer, A. Kotz Dittrich, D. Schmidt: Research Perspectives for Time Series Management Systems. SIGMOD RECORD, Vol. 23, No. 1, March 1994, pp. 10 - 15.

[5] W. Dreyer: Interoperability Issues in Time Series Management. Proceedings of the Workshop on Interoperability of Database Systems and Applications, pp. 235-246, Fribourg, Switzerland, October 1993.

[6] FAME Software Corporation: User's Guide to Fame, 1990.

[7] H. Hinterberger, J.C. French (eds.): Proc. of the 6th Intl. Working Conference on Scientific and Statistical Database Management, Ascona, Switzerland, 1992.

[8] K.E. Iverson: A Programming Language. Wiley, New York, 1962.

[9] K.E. Iverson: A Dictionary of APL. ACM Quote-Quad, Vol. 18, No. 4, 1987.

[10] C.S. Jensen (ed.) et al.: Proposed Temporal Database Concepts. Proc. of the Intl. Workshop on an Infrastructure for Temporal Databases, Arlington, TX, June 1993.

[11] Z. Michalewicz: Statistical and Scientific Databases. Ellis Horwood Ltd., 1991.

[12] A. Segev, R. Chandra: A Data Model for Time-Series Analysis. Workshop on Current Issues in Databases and Applications, Rutgers Univ., Oct 1992. In: Advanced Database Systems, editors: N. Adam and B. Bhargava, Lectures Notes in Computer Science Series, Springer Verlag, 1993.

[13] A. Segev, A. Shoshani: A Temporal Data Model Based on Time Sequences. In [15], chapter 11, pp. 248 - 269.

[14] M.D. Soo et al.: Architectural Extensions to Support Multiple Calendars. TEMPIS Technical Report, No.32, Univ. of Arizona, Dept. of Computer Science, 1992.

[15] A.U. Tansel et al.: Temporal Databases - Theory, Design and Implementation. The Benjamin/Cummings Publ. Comp., 1993.

[16] A. Weinand, E. Gamma, R. Marty: ET++ – An Object-Oriented Application Framework in C++. Structured Programming, Vol. 10, No. 2, June 1989.