The TEMPIS Project

Architectural Extensions to Support Multiple Calendars

Michael D. Soo, Richard Snodgrass,
Curtis E. Dyreson, Christian S. Jensen, and Nick Kline

Revised May, 1992

TEMPIS Technical Report No. 32

**Department of Computer Science
University of Arizona
Tucson, AZ 85721**

The TEMPIS Project

Architectural Extensions to Support Multiple Calendars

Michael D. Soo, Richard Snodgrass,
Curtis E. Dyreson, Christian S. Jensen, and Nick Kline

Revised May, 1992

Abstract

We describe in detail a system architecture for supporting a time-stamp attribute domain in conventional relational database management systems. This architecture underlies previously proposed temporal modifications to SQL. We describe the major components of the system and how they interact. For each component of the system, we provide specifications for the routines exported by that component. Finally, we describe a preliminary design for a toolkit that aids in the generation of the components of the database management system that support time.

TEMPIS Technical Report No. 32

**Department of Computer Science
University of Arizona
Tucson, AZ 85721**

Contents

1 Introduction

This paper is a detailed description of the design of the architectural extensions to a database management system (DBMS) supporting multiple calendars. The paper contains descriptions of the modules comprising the architectural extensions, with detailed descriptions of the services provided by each module. In addition, the data structures used by each system module are included.

It is assumed that the reader is familiar with two papers. First, the system architecture underlies query language modifications to SQL. The modified SQL supports calendric system selection, property table selection, and calendar independent temporal built-in functions, arithmetic operations, comparison operations and aggregate functions [Soo & Snodgrass 1992A]. Second, this paper concentrates on describing the details of the architecture, and it provides minimal motivation for the system design. The discussions motivating particular design decisions are the topic of a separate paper [Soo & Snodgrass 1992B].

The paper is organized as follows. In Section 2 we briefly summarize some of the query language features supported by the architecture. The next two sections contain an overview of the architecture and general comments on design decisions pertaining to all modules. Section 5 discusses the operations for time-stamp manipulation. Section 6 describes support for accessing calendar and calendric system provided services. The next section details the services provided by calendars. Section 8 details the module that makes possible the use of multilingual temporal constants. Section 9 contains examples illustrating how components in the architecture interact. A set of tools for generating a multiple calendar DBMS from high level specifications is described in Section 10. The appendix contains indexes for the function and data type definitions.

2 Example Language Constructs

Figure 1 shows an excerpt of an SQL module. The example illustrates many of the ways in which calendric systems and property tables may be specified; it is not intended to be realistic. A full description of the language features is provided elsewhere [Soo & Snodgrass 1992A].

```
...

declare calendric system as russian;

declare x cursor for
    select name, id, birthday,
        age with property_table_a, when_employed as american
    from employee
    where month_name_of(birthday) = 'jinvar' and
        birthday < |2 jinvar 1925| and
        age > %60 years% as american with property_table_b
        when_employed overlaps [1975];

procedure set_x_properties
    sqlcode
    set properties with x_property_table;

declare calendric system as american;

procedure open_x_cursor
    sqlcode
    open x;

...
```

Figure 1: Example of Calendric System and Property Selection

The `russian` calendric system is declared in the global scope. The scope of this declaration extends to the next global declaration, naming the `american` calendric system. The `russian` calendric system applies to the *birthday* and *age* attributes in the target list of the `select` statement since, unlike the *when_employed* attribute, no calendric system is locally declared for these attributes via an `as` clause. Similarly, the `russian` calendric system is used to resolve the function `month_name_of`, and to interpret the constants `|2 jinvar 1925|` and `1975` ("Jinvar" is a phonetic translation of the Russian word for "January"), while the `american` calendric system is used to interpret the span `%60 years%`. We note that, in this instance, the function `month_name_of` is defined via the `russian` calendric system and returns Russian month names.

In Figure 1, the procedure `set_x_properties` contains a command that activates the property values contained in the property table `x_property_table`. Invocation of this procedure causes the property values contained in that table to be activated. These values remain active until explicitly overridden by another `set properties` command. For example, if an application program calls the procedure `set_x_properties` prior to calling the procedure `open_x_cursor` then the property values specified in `x_property_table` override the property values in the default property table.

Conversely, naming a property table for an individual data item limits the activation of it's property values to the processing of that data item. For example, associated with the attribute `age` in the `select` clause is a property table `property_table_a`. The property values in this table are activated temporarily while time-stamps are being converted for the `age` attribute.

3 Architectural Overview

Figure 2 contains a diagram showing the major components of the system. Each box in the figure represents a component of the system; a solid line arrow from one component to another indicates that the former utilizes services provided by the latter. Data structures (non-procedural components) are shown as ovals, and a dashed line arrow indicates a data structure contains a reference to another component.

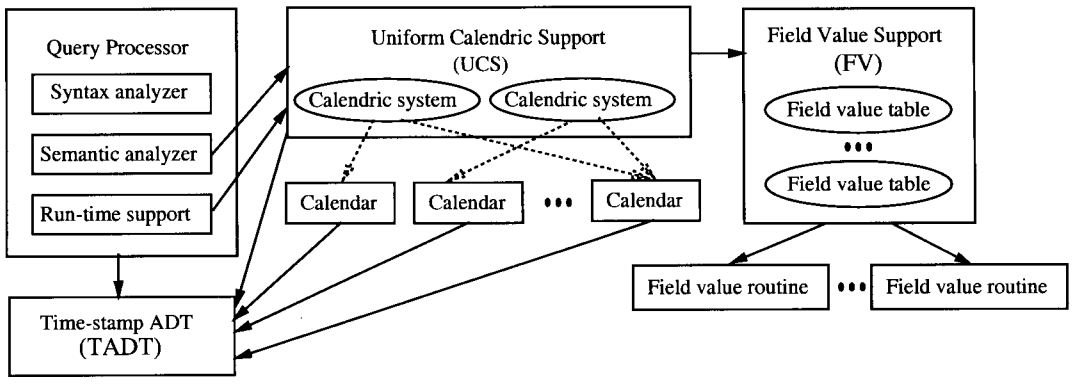

Figure 2: System Architecture Overview

The figure shows the following components.

- Query processor—a conventional query processing system extended to support the new temporal constructs.
- Uniform calendric support (UCS)—an interface that manages access to the services provided by calendars. Each calendric system is defined as a collection of data structures within the

UCS. Within the architecture, calendric systems have no procedural component; they merely provide a mechanism for accessing the services exported by their calendars.

- Field Value Support (FV)—a set of tables and routines supporting extensible formatting of temporal constants.
- Calendar—a set of tables and routines implementing calendar dependent operations. We note that, as shown in Figure 2, calendars can be shared by multiple calendric systems.
- Time-stamp ADT (TADT) support—a set of routines encapsulating operations on physical time-stamps. The TADT implements all temporal operations that do not require interpretation by a calendar.

In Figure 2, it is readily apparent that the support for calendar dependent operations is partitioned from the support for calendar independent operations. The UCS is responsible for executing, by using the appropriate calendric system, all calendar dependent operations. The TADT, on the other hand, provides calendar independent operations, specifically, memory management and built-in, arithmetic, comparison, and aggregate operations on time-stamps.

This distinction is the key aspect of our approach. We isolate operations requiring calendar interpretation by encapsulating them within a calendar, and provide calendar independent operations elsewhere. This allows the architecture to support extensibility and interchangeability of calendric systems and calendars.

As an example, consider the arithmetic operation of computing the sum of two span values [Soo & Snodgrass 1992A]. Variable spans require calendar interpretation while fixed spans do not. Therefore, the TADT exports an operation `tadt_fs_add_fs` which adds two fixed spans, while a specific calendar *calendar* must provide an operation for adding a variable span to a fixed span, `cal_`*calendar*`_vs_add_fs`, and an operation for adding a variable span to a variable span, `cal_`*calendar*`_vs_add_vs`. The UCS exports a generic span addition operation, `ucs_s_add_s`. The query processor invokes `ucs_s_add_s` whenever a span addition operation is performed. `ucs_s_add_s` uses the TADT operation `tadt_is_fs` to determine if its operands are variable or fixed spans, then calls the appropriate TADT or calendar routine. Section 9.2 elaborates on this example.

Extensibility of calendric systems and calendars is central to our architecture. We therefore support definition of calendric systems and calendars by local site personnel. A base version of the DBMS will likely include several calendars and calendric systems, and these calendars and calendric systems will be adequate for most users. In addition, we anticipate a market for customized calendars and calendric systems with third party vendors specializing in developing such solutions.

Calendar and calendric system definition will be performed by a *database implementor* (DBI), a person with sufficient knowledge of the internal workings of the database management system to implement calendar-defined functions and routines. The DBI is responsible for supplying the supporting components of calendars and calendric systems and generating the resulting database management system. To simplify this task, we provide a *calendar DBMS generation toolkit* that accepts calendar, calendric system, and field value specifications provided by the DBI and composes the DBMS from those specifications and preexisting components. The design of the generation toolkit is presented in Section 10.

We continue by describing the system components shown in Figure 2 in more detail. For each component in the figure, we describe the functions that comprise it. The functions are specified using their name, a description of their purpose, input and output parameters, return values, and error codes. Function signatures are given in ANSI C.

4 Global Design Decisions

This section discusses design decisions that are pertinent to every architectural module.

Time-stamps are stored in memory during processing. Memory to store a time-stamp can be allocated by request of the run-time system, the UCS, or the TADT. It is the responsibility of the module that requested memory for a time-stamp to free that memory when it is no longer needed. When memory is requested for a time-stamp, the request always demands the maximum possible time-stamp size because a time-stamp may change size as a result of an arithmetic operation. For example, the result of adding a high resolution event time-stamp (8 bytes) to a low resolution fixed span time-stamp (8 bytes), could be an extended resolution event time-stamp (12 bytes). By allocating 12 (or more) bytes initially, the result can be stored safely in memory. Although this memory allocation scheme wastes some space, there will be few time-stamps in memory at any time. Furthermore, it eliminates a run-time check to ensure that the target of a time-stamp assignment is spacious enough to receive the time-stamp.

Memory for aggregate structures is allocated by the aggregate initialization routines and freed by the run-time system. Calendars may only allocate memory for aggregates. All other calendar data structures must be static. This reduces the possibility of memory leaks caused by faulty calendar implementations. In general, however, we will assume that calendar implementations are well-behaved. For example, no UCS or TADT routine checks the validity of run-time data structures (e.g., that a time-stamp is actually an event) passed by a calendar implementation. The data structures are assumed to be valid.

Another system-wide design feature is a uniform and consistent error handling discipline. Every architectural module uses the same error handling strategy. Only some routines can detect run-time errors; those routines return a value of `error_type`. If a routine detects an error during processing, it immediately returns the appropriate error code to the callee. The callee is responsible for recovering from the error (or, if it cannot recover, passing the error up the calling stack). A routine that detects an error must leave all input and output parameters unchanged. If no error is detected, the routine returns an `ok` value (defined to be 0).

Error codes are either defined on a per component basis as part of the external data structures or defined globally as part of the `error_type`. Error codes which are germane to more than one component are globally defined. For example, the system has an `out_of_range` error code. This error code would be returned by a time-stamp arithmetic routine that detected that the result of a time-stamp arithmetic operation exceeded the maximum or minimum representable time. `out_of_range` is a global error since it could be returned by a calendar routine, the TADT, or the UCS. On the other hand, some TADT routines return a `tadt_error_type`. This type also includes a value `tadt_ok`, with the same value, 0, as `ok`.

The header comment for each routine that returns an error code indicates which error codes it can return. Every routine can return the `ok` value. We omit this fact from the routine header comments for brevity.

Below are system-wide data structures, including those for error handling.

```
typedef enum {
          ok, /* successful completion of the routine */
          out_of_range,
          divide_by_zero,
          overflow,
          undefined_extremum,
          conversion_error,
          insufficient_space,
          semantic_error,
          string_overflow
} error_type;
```

```
typedef enum { FALSE, TRUE} bool;


/*
*used to pass strings parameters
*/


typedef struct {
          int length;
          char *string;
} *string_struct;
```

5 Time-stamp ADT Support

As previously mentioned, the TADT is responsible for all temporal operations that do not require calendar interpretation. This includes all operations on event, interval, and fixed span values plus auxiliary operations for time-stamp manipulation. The representations of all such values as stored in the database are necessarily calendar independent [Dyreson & Snodgrass 1992]. Operations involving variable spans require calendar support and are not implemented by the TADT.

Table 1 shows the types of operations supported by the TADT along with a count of routines for each operation type.

Operation	*Number of Routines*
Built-in operations	9
Memory operations	10
Arithmetic operations	11
Comparison operations	13
Aggregate operations	30
Time-stamp creation and manipulation	12

Table 1: Time-stamp ADT Operations

As shown in Figure 2, the operations provided by the TADT are used by the run-time support of the query processor, the UCS, and any calendars defined in the system. The query processor invokes the TADT to allocate runtime data structures, as well as to execute all built-in, memory allocation, arithmetic, comparison, and aggregate operations involving non-span operands. A calendar calls the time-stamp creation routines of the TADT while computing a time-stamp equivalent to a temporal constant encountered in the input. The UCS invokes the TADT to execute fixed span operations, as we discuss in Section 6.

5.1 External Data Structures

Operations on event, interval, and fixed span values are operations on complex time-stamp data structures. However, the complexity of these data structures is hidden from the other architectural components, beneath a simple interface advertised by the TADT. The data structures presented below are part of this interface and are needed to facilitate efficient communication with the TADT.

```
typedef enum {
          tadt_ok,
          tadt_invalid_size, /*time-stamp space error*/
```

```
          tadt_invalid_interval, /*interval error*/
          tadt_truncation /*had to truncate time-stamp*/
} tadt_error_type;


/*
*The internal "space" datatypes are defined elsewhere
*/

typedef event_space_type *event_type;


typedef span_space_type *span_type;


typedef interval_space_type *interval_type;


typedef timestamp_space_type *timestamp_type;


typedef enum { hi, low, extended, special} resolution_type;


typedef enum { beginning, forever, undefined} special_value_type;


/*
*Unpacked time information structure
*/

typedef struct {
          resolution_type resolution;
          special_value_type special;
          unsigned char sign;
          unsigned short milli; /*max value is 1000*/
          unsigned short micro; /*max value is 1000*/
          unsigned short nano; /*max value is 1000*/
          unsigned int secondshi; /*max value is 2**27*/
          unsigned int secondslo; /*max value is 2**32*/
          unsigned int extra; /*we currently support only 1 extra word of precision*/
} seconds_space_type;


typedef seconds_space_type *seconds_type;


/*
*Unpacked variable span information structure
*/

typedef struct {
          short param1;
          short param2;
          short param3;
          unsigned char cal_id;
          unsigned char span_id;
} vs_space_type;
```

```
typedef vs_space_type *vs_type;


/*
*Aggregate structures
*/

typedef struct {
          int count;
          event_type sum_of_events;
} *avg_state_e;


typedef struct {
          int count;
          span_type sum_of_spans;
} *avg_state_fs;
```

5.2 Externally Visible Routines

In this section, we describe the routines comprising the TADT.

5.2.1 Memory Allocation

The TADT must allocate memory for new events, intervals, spans, and other data structures. Each TADT external data structure is allocated by a different routine. There is also a general allocation routine, `tadt_malloc`. Allocated memory must be de-allocated when it is not longer needed using the `tadt_free` routine. This memory area is managed very similarly to the standard Unix memory allocation routines. The memory management routines follow.

```
/*
*Routine:  tadt_malloc
*
*Description:  Allocate storage from the TADT memory heap.  Marks a region
*              of mem_size bytes which is now in use.  If no more memory
*              can be allocated, the behaviour of this function is undefined.
*
*Arguments:  size -- (IN): the number of bytes to allocate
*
*Return Value:  The address of allocated memory
*
*Errors:  None
*
*Side Effects:  Some of the TADT heap is marked as in use.  This may lead
*               to memory leaks later on.
*/
void *tadt_malloc(int size);


/*
*Routine:  tadt_free
*
*Description:  Frees previously allocated storage
*
*Arguments:  address -- (IN): address of previously allocated storage
*
*Return Value:  TRUE if the address was de-allocated, FALSE otherwise
*
*Errors:  None
*
*Side Effects:  If the address has not been allocated previously, then
*               this will could lead to internal errors.
*/
bool tadt_free(void *address);
```

```
/*
*Routine:  tadt_allocate_e
*
*Description:  Memory allocation for an event
*
*Arguments:  resolution -- (IN): which resolution to allocate
*
*Return Value:  Address of allocated event
*
*Errors:  None
*
*Side Effects:  Mallocs space
*/
event_type tadt_allocate_e(resolution_type resolution);


/*
*The remaining memory allocation routine prologues are omitted for brevity
*/
span_type tadt_allocate_s(resolution_type resolution);
interval_type tadt_allocate_i(resolution_type resolution1, resolution_type resolution2);
timestamp_type tadt_allocate_ts();
vs_type tadt_allocate_vs_struct();
avg_state_e tadt_allocate_avg_state_e();
avg_state_fs tadt_allocate_avg_state_fs();
string_struct tadt_allocate_string_struct();
```

5.2.2 Built-in Function Support

The built-in function component of the TADT provides support for the built-in query language functions.

```
/*
*Routine:  tadt_begin
*
*Description:  Returns the event delimiting the beginning of the interval
*
*Arguments:  interval -- (IN): the input interval
*            result -- (OUT): result event
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
tadt_error_type tadt_begin(interval_type interval, event_type result);



/*
*Routine:  tadt_end
*
*Description:  Returns the event delimiting the end of the interval
*
*Arguments:  interval -- (IN): the input interval
*            result -- (OUT): result event
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
tadt_error_type tadt_end(interval_type interval, event_type result);
```

```
/*
*Routine:  tadt_interval
*
*Description:  Constructs an interval from two input events,
*              if event1 >= event2 then it flags an error
*
*Arguments:  event1 -- (IN): the beginning of the interval
*            event2 -- (IN): the end of the interval
*            result -- (OUT): result interval
*
*Return Value:  Error code
*
*Errors:  tadt_invalid_interval
*
*Side Effects:  assignment to passed result structure
*/
tadt_error_type tadt_interval(event_type event1, event_type event2, interval_type result);



/*
*Routine:  tadt_intersect
*
*Description:  Returns the overlap of two intervals, if
*              they intersect.  Otherwise leave the result
*              unchanged.
*
*Arguments:  i1, i2 -- (IN): intervals for intersection
*            result -- (OUT): result interval if intervals intersect
*
*Return Value:  Error code
*
*Errors:  tadt_invalid_interval
*
*Side Effects:  Assignment to passed result structure
*/
tadt_error_type tadt_intersect(interval_type i1, interval_type i2, interval_type result);



/*
*Routine:  tadt_span
*
*Description:  Returns the span of an input interval, a positive span
*
*Arguments:  interval -- (IN): the input interval
*            result -- (OUT): result span
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
tadt_error_type tadt_span(interval_type interval, span_type result);



/*
*Routine:  tadt_abs_fs
*
*Description:  Returns the absolute value of a fixed span
*
*Arguments:  span -- (IN): input span
*            result -- (OUT): result positive span
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
tadt_error_type tadt_abs_fs(span_type span, span_type result);
```

```
/*
*Routine:  tadt_first
*
*Description:  Returns the earlier of two events
*
*Arguments:  event1, event2 -- (IN): two input events
*            result -- (OUT) earliest event.
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
tadt_error_type tadt_first(event_type event1, event_type event2, event_type result);



/*
*Routine:  tadt_last
*
*Description:  Returns the later event of two events
*
*Arguments:  event1, event2 -- (IN): two input events
*            result -- (OUT): latest event.
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
tadt_error_type tadt_last(event_type event1, event_type event2, event_type result);



/*
*Routine:  tadt_present
*
*Description:  Returns the present time
*
*Arguments:  result -- (OUT): present event
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
tadt_error_type tadt_present(event_type result);
```

5.2.3 Arithmetic Operations Support

The arithmetic component of the TADT is shown below. A single routine is provided for commutative operations such as addition and multiplication.

The names of dyadic arithmetic functions have the form `tadt_`*operand1*`_`*operation*`_`*operand2* where *operand1* denotes the type of the first operand, *operation* denotes the arithmetic operation being performed, and *operand2* denotes the type of the second operand. Monadic functions follow an analogous naming convention.

```
/*
*Routine:  tadt_neg_fs
*
*Description:  Determine the sign opposite fixed span which is otherwise the
*              same as the input span.  Has no effect on a zero length
*              span.
*
```

```
*Arguments:  span -- (IN) : span to be converted
*            result -- (OUT) : result span
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_neg_fs(span_type span, span_type result);



/*
*Routine:  tadt_fs_add_fs
*
*Description:  Add two fixed spans
*
*Arguments:  span1, span2 -- (IN) : spans to be added
*            result -- (OUT) : result span
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_fs_add_fs(span_type span1, span_type span2, span_type result);



/*
*Routine:  tadt_fs_minus_fs
*
*Description:  Subtract two fixed spans
*
*Arguments:  span1 -- (IN) : minuend (span to be subtracted from)
*            span2 -- (IN) : subtrahend
*            result -- (OUT) : result span
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_fs_minus_fs(span_type span1, span_type span2, span_type result);



/*
*Routine:  tadt_e_add_fs
*
*Description:  Displace an event by adding a span
*
*Arguments:  event -- (IN) : the event time
*            span -- (IN) : the displacement
*            result -- (OUT) : result event
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_e_add_fs(event_type event, span_type span, event_type result);



/*
*Routine:  tadt_e_minus_fs
*
```

```
*Description:  Displace an event by subtracting a span
*
*Arguments:  event -- (IN) : the event time
*            span -- (IN) : the displacement
*            result -- (OUT) : result event
*
*Return Value:  Error code
*
*Errors:  out_of_range
*         overflow (beginning - anything)
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_e_minus_fs(event_type event, span_type span, event_type result);



/*
*Routine:  tadt_e_minus_e
*
*Description:  Compute the span between two events by
*              subtracting the second event from the first event (e1 - e2).
*              if e1 > e2, return a positive span
*              if e1 < e2, return a negative span
*
*Arguments:  event1, event2 -- (IN) : two input events.
*            result -- (OUT) : result span
*
*Return Value:  Error code
*
*Errors:  overflow (forever - beginning)
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_e_minus_e(event_type event1, event_type event2, span_type result);



/*
*Routine:  tadt_fs_times_n
*
*Description:  Return a span equal to a multiple of the input span.  Although
*              the multiplier may be a floating point number, the result is
*              always an integral number of chronons.
*
*Arguments:  span -- (IN): the input span
*            n -- (IN): number of times
*            result -- (OUT) : result span
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_fs_times_n(span_type span, double n, span_type result);



/*
*Routine:  tadt_fs_div_n
*
*Description:  Divide a span by a numeric value.  N must be nonzero.
*
*Arguments:  span -- (IN):the input span
*            n -- (IN): number of division
*            result -- (OUT) : result span
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
```

```
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_fs_div_n(span_type span, double n, span_type result);



/*
*Routine:  tadt_fs_div_fs
*
*Description:  Divide one span by another
*
*Arguments:  span1 -- (IN): the dividend
*            span2 --(IN): the divisor
*            result -- (OUT) : result integer
*
*Return Value:  Error code
*
*Errors:  out_of_range
*         division_by_zero
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_fs_div_fs(span_type span1, span_type span2, double *result);



/*
*Routine:  tadt_i_add_fs
*
*Description:  Displace an interval by adding a span
*
*Arguments:  interval -- (IN): the interval to be displaced
*            span -- (IN): the span of displacement
*            result -- (OUT) : result interval
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_i_add_fs(interval_type interval, span_type span, interval_type result);



/*
*Routine:  tadt_i_minus_fs
*
*Description:  Displace an interval by subtracting a span
*
*Arguments:  interval -- (IN): the interval to be displaced
*            span -- (IN): the span of displacement
*            result -- (OUT) : result interval
*
*Return Value:  Error code
*
*Errors:  out_of_range
*         overflow (|beginning,forever| - anything)
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_i_minus_fs(interval_type interval, span_type span, interval_type result);
```

5.2.4 Comparison Operations Support

The semantics of the time-stamp comparison operations depends on the underlying temporal model [Dyreson & Snodgrass 1992]. Our underlying model of time is a *discrete* model where *chronons* are

nondecomposable units of time. A time-stamp records that an event occurred *during* a particular chronon. One consequence of this model is that two events which occur during the same chronon may still occur at different times.

We do, however, have a comparison operations such as an equality test for event time-stamps. The equality test simply checks to see if the events occurred during the same chronon. In fact, all comparison operations are operations that compare "chronons" rather than "times". Hence, the equality comparison does not establish whether or not two events occur at the same time, only whether they occur during the same chronon. The test captures the intuitive comparison semantics without compromising the theoretical considerations.

The comparison operations component of the TADT is shown below. As with arithmetic operations, a single routine is provided for commutative operations. The naming conventions for comparison routines are similar to the arithmetic functions.

```
/*
*Routine:  tadt_fs_eq_fs
*
*Description:  This routine compares two fixed span values.  It returns
*              the boolean value TRUE if they are equal and FALSE otherwise.
*
*Arguments:  span1, span2 -- (IN) : spans to be compared
*
*Return Value:  TRUE if span1 == span2, FALSE otherwise
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_fs_eq_fs(span_type span1, span_type span2);



/*
*Routine:  tadt_fs_lt_fs
*
*Description:  This routine compares two fixed span values.  It returns
*              the boolean value TRUE if the first span is less than
*              the second span and FALSE otherwise.
*
*Arguments:  span1, span2 -- (IN) : spans to be compared
*
*Return Value:  TRUE if span1 < span2, FALSE otherwise
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_fs_lt_fs(span_type span1, span_type span2);



/*
*Routine:  tadt_fs_gt_fs
*
*Description:  This routine compares two fixed span values.  It returns
*              the boolean value TRUE if the first span is greater than
*              the second span and FALSE otherwise.
*
*Arguments:  span1, span2 -- (IN) : spans to be compared
*
*Return Value:  TRUE if span1 > span2, FALSE if span1 =< span2
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_fs_gt_fs(span_type span1, span_type span2);
```

```
/*
*Routine:  tadt_e_eq_e
*
*Description:  This routine compares two event values.  It returns
*              the boolean value TRUE if the events are equal and
*              FALSE otherwise.  Two events are defined to be
*              equal if they occur during the same chronon,
*              not if they occur at the same time.
*              The size of a chronon is resolution dependent.
*
*Arguments:  event1, event2 -- (IN) : events to be compared
*
*Return Value:  TRUE if event1 = event2, FALSE if event1 <> event2
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_e_equals_e(event_type event1, event_type event2);



/*
*Routine:  tadt_e_precedes_e
*
*Description:  This routine compares two event values.  It returns
*              the boolean value TRUE if the first event precedes the
*              second event and FALSE otherwise.
*
*Arguments:  event1, event2 -- (IN) : events to be compared
*
*Return Value:  TRUE if event1 < event2, FALSE if event1 >= event2
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_e_precedes_e(event_type event1, event_type event2);



/*
*Routine:  tadt_e_precedes_i
*
*Description:  This routine compares an event and an interval.  It returns
*              the boolean value TRUE if the event precedes the
*              beginning of the interval and FALSE otherwise.
*
*Arguments:  event, interval -- (IN) : event and interval to be compared
*
*Return Value:  TRUE if event < begin(interval),
*               FALSE if event >= begin(interval)
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_e_precedes_i(event_type event, interval_type interval);



/*
*Routine:  tadt_e_overlaps_i
*
*Description:  This routine compares an event and an interval.  It returns
*              the boolean value TRUE if the event overlaps the
*              interval and FALSE otherwise.
*
*Arguments:  event, interval -- (IN) : event and interval to be compared
*
*Return Value:  TRUE if begin(interval) <= event <= end(interval)
*               FALSE if event < begin(interval) or event > end(interval)
*
```

```
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_e_overlaps_i(event_type event, interval_type interval);



/*
*Routine:  tadt_i_precedes_e
*
*Description:  This routine compares an interval and an event.  It returns
*              the boolean value TRUE if the interval precedes the
*              event and FALSE otherwise.
*
*Arguments:  interval, event -- (IN) : interval and event to be compared
*
*Return Value:  TRUE if end(interval) < event
*               FALSE if event <= end(interval)
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_i_precedes_e(interval_type interval, event_type event);



/*
*Routine:  tadt_i_precedes_i
*
*Description:  This routine compares two intervals.  It returns
*              the boolean value TRUE if the end of the first interval
*              precedes the beginning of the second.
*
*Arguments:  interval1, interval2 -- (IN) : intervals to be compared
*
*Return Value:  TRUE if end(interval1) < begin(interval2)
*               FALSE if end(interval1) >= begin(interval2)
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_i_precedes_i(interval_type interval1, interval_type interval2);



/*
*Routine:  tadt_i_eq_i
*
*Description:  This routine compares two intervals.  It returns
*              the boolean value TRUE if the two intervals are
*              equal and FALSE otherwise.
*
*Arguments:  interval1, interval2 -- (IN) : intervals to be compared
*
*Return Value:  bool TRUE if interval1 = interval2
*               bool FALSE if interval1 <> interval2
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_i_equals_i(interval_type interval1, interval_type interval2);



/*
*Routine:  tadt_i_meets_i
*
*Description:  This routine compares two intervals.  It returns
```

```
*              the boolean value TRUE if the end of the first interval
*              is the beginning of the second and FALSE otherwise.
*
*Arguments:  interval1, interval2 -- (IN) : intervals to be compared
*
*Return Value:  TRUE if end(interval1) = begin(interval2)
*               FALSE if end(interval1) <> begin(interval2)
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_i_meets_i(interval_type interval1, interval_type interval2);



/*
*Routine:  tadt_i_overlaps_i
*
*Description:  This routine compares two intervals.  It returns
*              the boolean value TRUE if the intervals share at least
*              one chronon in common and FALSE otherwise.
*
*Arguments:  interval1, interval2 -- (IN) : intervals to be compared
*
*Return Value:  TRUE if interval1 overlaps interval2
*               FALSE if interval1 doesn't overlap interval2
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_i_overlaps_i(interval_type interval1, interval_type interval2);



/*
*Routine:  tadt_i_contains_i
*
*Description:  This routine compares two intervals.  It returns
*              the boolean value TRUE if the first interval contains
*              the second interval and FALSE otherwise.
*
*Arguments:  interval1, interval2 -- (IN) : intervals to be compared
*
*Return Value:  TRUE if interval1 contains interval2
*               FALSE if interval1 doesn't contain interval2
*
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_i_contains_i(interval_type interval1, interval_type interval2);
```

5.2.5 Aggregate Function Support

The TADT supports aggregate computation over events, intervals, and fixed spans. Calendars are responsible for aggregate computations involving variable spans.

A stream model is used for aggregate computation. Three routines, an initialization routine, an accumulation routine, and a final result routine, are defined for each aggregate function and each data type. Data structures that accumulate the results of an aggregate computation are allocated by initialization routines. The accumulation routines store the partial result of an aggregate function in this structure, and the final result routines compute the result of the aggregate from the accumulated information.

To support the aggregate functions `min` and `max`, there is a special *undefined* value for events and spans. This value means that the time-stamp is uninitialized.

A name of an aggregate support function has the form `tadt_`*phase*`_`*aggregate*`_`*type* where *phase* denotes either the `init`, `accum`, or `final` processing stage, *aggregate* denotes the name of the aggregate function, and *type* denotes the event, span, or interval data type over which aggregate function is being computed. We only show routine prologues for the initialization, accumulation, and final stage of one aggregate function. The remaining prologues are similar.

```
/*
*Routine:  tadt_init_count_e
*
*Description:  Allocate and initialize the integer used for the event
*              count aggregate and return a pointer to it
*
*Arguments:  None
*
*Return Value:  int *:  allocated and initialized with 0 for computation
*
*Errors:  None
*
*Side Effects:  Allocates memory
*/
int *tadt_init_count_e();


/*
*Routine:  tadt_accum_count_e
*
*Description:  Accumulate the next event in the stream
*
*Arguments:  count_state -- (IN&OUT) : intermediate count state
*            event -- (IN) : the next event
*
*Return Value:  Error code
*
*Errors:  overflow
*
*Side Effects:  Overwrites count state
*/
error_type tadt_accum_count_e(int *count_state, event_type event);


/*
*Routine:  tadt_final_count_e
*
*Description:  Return the final count of events
*
*Arguments:  final_state -- (IN) : the final state contains the count
*            final_count -- (OUT): final count
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Overwrites final_count
*/
error_type tadt_final_count_e(int final_state, int *final_count);


avg_state_e tadt_init_avg_e();
error_type tadt_accum_avg_e(avg_state_e avg_state, event_type event);
error_type tadt_final_avg_e(avg_state_e final_state, event_type event);


event_type tadt_init_max_e();
error_type tadt_accum_max_e(event_type max_state, event_type event);
error_type tadt_final_max_e(event_type final_state, event_type event);
```

```
event_type tadt_init_min_e();
error_type tadt_accum_min_e(event_type min_state, event_type event);
error_type tadt_final_min_e(event_type final_state, event_type event);


int *tadt_init_count_i();
error_type tadt_accum_count_i(int *count_state, interval_type interval);
error_type tadt_final_count_i(int final_state, int *final_count);


int *tadt_init_count_fs();
error_type tadt_accum_count_fs(int *count_state, span_type span);
error_type tadt_final_count_fs(int final_state, int *final_count);


span_type tadt_init_sum_fs();
error_type tadt_accum_sum_fs(span_type sum_state, span_type span);
error_type tadt_final_sum_fs(span_type final_state, span_type span);


avg_state_fs tadt_init_avg_fs();
error_type tadt_accum_avg_fs(avg_state_fs avg_state, span_type span);
error_type tadt_final_avg_fs(avg_state_fs final_state, span_type span);


span_type tadt_init_max_fs();
error_type tadt_accum_max_fs(span_type max_state, span_type span);
error_type tadt_final_max_fs(span_type final_state, span_type span);


span_type tadt_init_min_fs();
error_type tadt_accum_min_fs(span_type min_state, span_type span);
error_type tadt_final_min_fs(span_type final_state, span_type span);
```

5.2.6 Time-stamp Creation and Manipulation

The time-stamp creation routines provided by the TADT are shown below. These routines are invoked by a calendar when translating temporal constants to time-stamps.

```
/*
*Routine:  tadt_create_e
*
*Description:  Builds an event timestamp from the given chronon information.
*
*
*Arguments:  seconds -- (IN): seconds structure
*            resolution -- (IN): resolution
*            result -- (OUT): result event
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_create_e(seconds_type seconds, resolution_type resolution, timestamp_type result);



/*
*Routine:  tadt_create_i
*
*Description:  Builds an interval timestamp from the given events
*
*Arguments:  event1 -- (IN): starting event
*            event2 -- (IN): terminating event
```

```
*            result -- (OUT): result interval
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_create_i(event_type event1, event_type event2, timestamp_type result);



/*
*Routine:  tadt_create_fs
*
*Description:  Builds a fixed span from the given chronon information.
*
*Arguments:  seconds -- (IN): structure containing span info
*            resolution -- (IN): resolution
*            result -- (OUT): result fixed span
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_create_fs(seconds_type seconds, resolution_type resolution, timestamp_type result);



/*
*Routine:  tadt_create_vs
*
*Description:  Builds a variable span from the given information.
*
*Arguments:  vs -- (IN): vspan structure (contains vspan info)
*            result -- (OUT): resulting variable span
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_create_vs(vs_type vs, span_type result);



/*
*Routine:  tadt_extract_vs
*
*Description:  Extract the variable span information from the given vspan.
*
*Arguments:  vs -- (IN): variable span time-stamp
*            result -- (OUT): resulting variable span information structure
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed result structure
*/
error_type tadt_extract_vs(span_type vs, vs_type result);



/*
*Routine:  tadt_is_vs
*
*Description:  Determine if a span is a variable span.
*
```

```
*Arguments:  span -- (IN): the timestamp to be tested
*
*Return Value:  TRUE if span is a variable span
*               FALSE otherwise
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_is_vs(span_type span);



/*
*Routine:  tadt_can_fit_ts
*
*Description:  Determine if there is enough space allocated to
*              store a time-stamp.  Time-stamp sizes can be
*              determined using the sizeof function.
*
*Arguments:  ts -- (IN): the timestamp to be tested
*            size -- (IN): how much space is available
*
*Return Value:  TRUE if there is enough space
*               FALSE otherwise
*Errors:  None
*
*Side Effects:  None
*/
bool tadt_can_fit_ts(timestamp_type ts, int size);



/*
*Routine:  tadt_coerce_e
*
*Description:  Fit an event time-stamp into a certain space, truncating if
*              needed.  An extended resolution event time-stamp must be
*              coerced if it is to fit in the space of a high or
*              low resolution time-stamp.  If the extended resolution
*              is in the high resolution range, then it is coerced
*              to a high resolution, otherwise it is coerced to a
*              low resolution with truncation flagged.  Time-stamp sizes
*              can be determined using the sizeof function.  Only valid
*              event time-stamp sizes are allowed.  Other sizes will
*              generate an error.
*
*Arguments:  event -- (IN): the timestamp to be tested
*            size -- (IN): how much space is available
*            result -- (OUT): the coerced result
*
*Return Value:  Error code
*
*Errors:  tadt_truncation
*         tadt_invalid_size
*
*Side Effects:  Assignment to passed result timestamp
*/
tadt_error_type tadt_coerce_e(event_type event, int size, event_type result);



/*
*Routine:  tadt_coerce_i
*
*Description:  Fit an interval time-stamp into a certain space, truncating if
*              needed.  An extended resolution interval time-stamp must be
*              coerced if it is to fit in the space of a high or
*              low resolution time-stamp.  If the extended resolution
*              is in the high resolution range, then it is coerced
*              to a high resolution, otherwise it is coerced to a
*              low resolution with truncation flagged.  Time-stamp sizes
*              can be determined using the sizeof function.  Only valid
*              interval time-stamp sizes are allowed.  Other sizes will
```

```
*              generate an error.
*
*Arguments:  interval -- (IN): the timestamp to be tested
*            size -- (IN): how much space is available
*            result -- (OUT): the coerced result
*
*Return Value:  Error code
*
*Errors:  tadt_truncation
*         tadt_invalid_size
*
*Side Effects:  Assignment to passed result timestamp
*/
tadt_error_type tadt_coerce_i(interval_type interval, int size, interval_type result);



/*
*Routine:  tadt_coerce_fs
*
*Description:  Fit a fixed span time-stamp into a certain space, truncating
*              if needed.  An extended resolution span time-stamp must be
*              coerced if it is to fit in the space of a high or
*              low resolution time-stamp.  If the extended resolution
*              is in the high resolution range, then it is coerced
*              to a high resolution, otherwise it is coerced to a
*              low resolution with truncation flagged.  Time-stamp sizes
*              can be determined using the sizeof function.  Only valid
*              fixed span time-stamp sizes are allowed.  Other sizes will
*              generate an error.
*
*Arguments:  span -- (IN): the timestamp to be tested
*            size -- (IN): how much space is available
*            result -- (OUT): the coerced result
*
*Return Value:  Error code
*
*Errors:  tadt_truncation
*         tadt_invalid_size
*
*Side Effects:  Assignment to passed result timestamp
*/
tadt_error_type tadt_coerce_fs(span_type span, int size, span_type result);



/*
*Routine:  tadt_e_to_seconds
*
*Description:  Break an event up into its unpacked form (seconds structure)
*
*Arguments:  e -- (IN): the event to be converted
*            seconds -- (OUT): the resulting seconds structure
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Assignment to passed seconds structure
*/
error_type tadt_e_to_seconds(event_type event, seconds_type seconds);



/*
*Routine:  tadt_seconds_to_e
*
*Description:  Convert an unpacked event into its packed form
*
*Arguments:  seconds -- (IN): the seconds structure to be converted
*            e -- (OUT): the resulting event
*
*Return Value:  Error code
```

```
*
*Errors:  None
*
*Side Effects:  Assignment to passed seconds structure
*/
error_type tadt_seconds_to_e(event_type event, seconds_type seconds);
```

5.3 Internal Data Structures

The time-stamp data structures that are accessible only within the TADT are given below. The interpretation of these data structures is presented elsewhere [Dyreson & Snodgrass 1992].

```
/*
*STANDARD EVENT TIMESTAMPS:
*/


typedef struct {
          unsigned ts_type    :   4;  /* type 0000 */
          unsigned wasted     :   1;  /* wasted */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   26; /* seconds since origin */
                                      /* word boundary */
          unsigned seconds2   :   12; /* seconds since origin */
          unsigned milli      :   10; /* milliseconds since origin */
          unsigned micro      :   10; /* microseconds since origin */
} hires_type;



typedef struct {
          unsigned ts_type    :   4;  /* type 0100 */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   27; /* seconds since origin */
                                      /* word boundary */
          unsigned seconds2   :   32; /* seconds since origin */
} lowres_type;



typedef struct {
          unsigned ts_type    :   4;  /* type 1000 */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   27; /* seconds since origin */
                                      /* word boundary */
          unsigned seconds2   :   32; /* seconds since origin */
                                      /* word boundary */
          unsigned milli      :   10; /* milliseconds since origin */
          unsigned micro      :   10; /* microseconds since origin */
          unsigned nano       :   10; /* nanoseconds since origin */
          unsigned wasted     :   1;  /* wasted */
          unsigned extraflag  :   1;  /* flag for extra word, more precision*/
} exres_type;



typedef struct {
          unsigned ts_type    :   4;  /* type 1100 */
          unsigned wasted     :   28; /* wasted */
                                      /* word boundary */
          unsigned special    :   32; /* which special event */
} special_type;
```

```
/*
*VARIABLE SPAN FORMAT
*/


typedef struct {
          unsigned ts_type    :   4;  /* type 1111 indicates variable span */
          unsigned cal_sys    :   5;  /* which calendric system */
          unsigned vspan      :   7;  /* which variable span function */
          unsigned param1     :   16; /* span can have three parameters */
                                      /* word boundary */
          unsigned param2     :   16; /* span can have three parameters */
          unsigned param3     :   16; /* span can have three parameters */
} vspan_type;



/*
*TIME-STAMP STORAGE STRUCTURES
*/


typedef union {
          hires_type hires;
          lowres_type lowres;
          exres_type exres;
          special_type special;
          hires_uni_chunked_type hires_uni_chunked;
          lowres_uni_chunked_type lowres_uni_chunked;
          exres_uni_chunked_type exres_uni_chunked;
          hires_nuni_chunked_type hires_nuni_chunked;
          lowres_nuni_chunked_type lowres_nuni_chunked;
          exres_nuni_chunked_type exres_nuni_chunked;
          hires_nuni_type hires_nuni;
          lowres_nuni_type lowres_nuni;
          exres_nuni_type exres_nuni;
} event_space_type;



typedef union {
          vspan_type vspan;
          event_space_type span;
} span_space_type;



typedef struct {
          event_space_type start;
          event_space_type terminate;
} interval_space_type;



typedef union {
          span_space_type span;
          event_space_type event;
          interval_space_type interval;
} timestamp_space_type;
```

The indeterminate time-stamp data structures are not fully supported by the routines (although the routines as defined will accept indeterminate values and may return indeterminate values). We include these data structures for completeness.

```
/*
*UNIFORM, CHUNKED INDETERMINATE FORMS (EVENTS):
*/
```

```
typedef struct {
          unsigned ts_type    :   4;  /* type 0001 */
          unsigned chunksize  :   4;  /* size of chunk */
          unsigned chunks     :   7;  /* number of chunks */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   16; /* seconds since origin */
                                      /* word boundary */
          unsigned seconds2   :   22; /* seconds since origin */
          unsigned milli      :   10; /* milliseconds since origin */
} hires_uni_chunked_type;



typedef struct {
          unsigned ts_type    :   4;  /* type 0101 */
          unsigned chunksize  :   4;  /* size of chunk */
          unsigned chunks     :   7;  /* number of chunks */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   16; /* 2048 seconds chunks since origin */
                                      /* word boundary */
          unsigned seconds2   :   32; /* seconds since origin */
} lowres_uni_chunked_type;



typedef struct {
          unsigned ts_type    :   4;  /* type 1001 */
          unsigned chunksize  :   4;  /* size of chunk */
          unsigned chunks     :   7;  /* number of chunks */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   16; /* seconds since origin */
                                      /* word boundary */
          unsigned seconds2   :   32; /* seconds since origin (cont) */
                                      /* word boundary */
          unsigned seconds3   :   11; /* seconds since origin (cont) */
          unsigned milli      :   10; /* milliseconds since origin */
          unsigned micro      :   10; /* microseconds since origin */
          unsigned extraflag  :   1;  /* flag for extra word, more precision*/
} exres_uni_chunked_type;



/*
*NONUNIFORM, CHUNKED INDETERMINATE FORMS (EVENTS):
*/


typedef struct {
          unsigned ts_type    :   4;  /* type 0010 */
          unsigned wasted     :   1;  /* wasted */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   26; /* seconds since origin */
                                      /* word boundary */
          unsigned seconds2   :   12; /* seconds since origin (cont) */
          unsigned milli      :   10; /* milliseconds since origin */
          unsigned micro      :   10; /* microseconds since origin */
                                      /* word boundary */
          unsigned chunksize  :   4;  /* size of chunk */
          unsigned chunks     :   7;  /* number of chunks */
          unsigned dist       :   7;  /* normalized distribution */
          unsigned rght_off   :   7;  /* right offset */
          unsigned left_off   :   7;  /* left offset */
} hires_nuni_chunked_type;
```

```
typedef struct {
          unsigned ts_type    :   4;  /* type 0110 */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   27; /* seconds since origin */
                                      /* word boundary */
          unsigned seconds2   :   32; /* seconds since origin (cont) */
                                      /* word boundary */
          unsigned chunksize  :   4;  /* size of chunk */
          unsigned chunks     :   7;  /* number of chunks */
          unsigned dist       :   7;  /* normalized distribution */
          unsigned rght_off   :   7;  /* right offset */
          unsigned left_off   :   7;  /* left offset */
} lowres_nuni_chunked_type;



typedef struct {
          unsigned ts_type    :   4;  /* type 1010 */
          unsigned sign       :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned seconds1   :   27; /* seconds since origin */
                                      /* word boundary */
          unsigned seconds2   :   32; /* seconds since origin (cont) */
                                      /* word boundary */
          unsigned milli      :   10; /* milliseconds since origin */
          unsigned micro      :   10; /* microseconds since origin */
          unsigned nano       :   10; /* nanoseconds since origin */
          unsigned wasted     :   1;  /* wasted */
          unsigned extraflag  :   1;  /* flag for extra word, more precision*/
                                      /* word boundary */
          unsigned chunksize  :   4;  /* size of chunk */
          unsigned chunks     :   7;  /* number of chunks */
          unsigned dist       :   7;  /* normalized distribution */
          unsigned rght_off   :   7;  /* right offset */
          unsigned left_off   :   7;  /* left offset */
} exres_nuni_chunked_type;



/*
*NONCHUNKED, NONUNIFORM INDETERMINATE FORMS (EVENTS):
*/


typedef struct {
          unsigned ts_type    :   4;  /* type 0011 */
          unsigned hiwasted   :   1;  /* wasted */
          unsigned hisign     :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned hiseconds1:    26; /* seconds since origin */
                                      /* word boundary */
          unsigned hiseconds2:    12; /* seconds since origin (cont) */
          unsigned himilli    :   10; /* milliseconds since origin */
          unsigned himicro    :   10; /* microseconds since origin */
                                      /* word boundary */
          unsigned lowasted   :   5;  /* wasted */
          unsigned losign     :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned loseconds1:    26; /* seconds since origin */
                                      /* word boundary */
          unsigned loseconds2:    12; /* seconds since origin (cont) */
          unsigned lomilli    :   10; /* milliseconds since origin */
          unsigned lomicro    :   10; /* microseconds since origin */
                                      /* word boundary */
          unsigned wasted     :   11; /* wasted */
          unsigned dist       :   7;  /* normalized distribution */
          unsigned rght_off   :   7;  /* right offset */
          unsigned left_off   :   7;  /* left offset */
} hires_nuni_type;
```

```
typedef struct {
          unsigned ts_type    :   4;  /* type 0111 */
          unsigned hisign     :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned hiseconds1:   27; /* seconds since origin */
                                     /* word boundary */
          unsigned hiseconds2:   32; /* seconds since origin (cont) */
                                     /* word boundary */
          unsigned lowasted   :   4;  /* wasted */
          unsigned losign2    :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned loseconds1:   27; /* seconds since origin */
                                     /* word boundary */
          unsigned loseconds2:   32; /* seconds since origin (cont) */
                                     /* word boundary */
          unsigned wasted     :   11; /* wasted */
          unsigned dist       :   7;  /* normalized distribution */
          unsigned rght_off   :   7;  /* right offset */
          unsigned left_off   :   7;  /* left offset */
} lowres_nuni_type;



typedef struct {
          unsigned ts_type    :   4;  /* type 1011 */
          unsigned hisign     :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned hiseconds1:   27; /* seconds since origin */
                                     /* word boundary */
          unsigned hiseconds2:   32; /* seconds since origin (cont) */
                                     /* word boundary */
          unsigned himilli    :   10; /* milliseconds since origin */
          unsigned himicro    :   10; /* microseconds since origin */
          unsigned hinano     :   10; /* nanoseconds since origin */
          unsigned hiwasted   :   1;  /* wasted */
          unsigned hiextraflag   1;  /* flag for extra word, more precision*/
                                     /* word boundary */
          unsigned lowasted   :   4;  /* wasted */
          unsigned losign     :   1;  /* sign flag -- 0 positive 1 negative */
          unsigned loseconds1:   27; /* seconds since origin */
                                     /* word boundary */
          unsigned loseconds2:   32; /* seconds since origin (cont) */
                                     /* word boundary */
          unsigned lomilli    :   10; /* milliseconds since origin */
          unsigned lomicro    :   10; /* microseconds since origin */
          unsigned lonano     :   10; /* nanoseconds since origin */
          unsigned lowasted2  :   1;  /* wasted */
          unsigned loextraflag   1;  /* flag for extra word, more precision*/
                                     /* word boundary */
          unsigned wasted     :   11; /* wasted */
          unsigned dist       :   7;  /* normalized distribution */
          unsigned rght_off   :   7;  /* right offset */
          unsigned left_off   :   7;  /* left offset */
} exres_nuni_type;



/*
*Probability Distribution Function Format
*/


typedef struct {
          unsigned wasted     :   10; /* wasted */
          unsigned dist       :   8;  /* normalized distribution */
          unsigned rght_off   :   7;  /* right offset */
          unsigned left_off   :   7;  /* left offset */
} prob_dist_type;
```

6 Uniform Calendric Support

The UCS provides a generic interface to all calendar defined services. Table 2 lists the types of operations performed by the UCS. The UCS is invoked by the query processor to activate and deactivate calendric systems and property values, convert temporal constants to time-stamps, convert time-stamps to temporal constants, resolve calendar defined functions, and execute all span operations. It maintains data structures defining calendric systems, and invokes calendar operations on behalf of the query processor. In general, the UCS is responsible for executing any operation which could possibly be calendar dependent.

Operation	*Number of Routines*
Memory allocation	4
Span arithmetic operations	10
Span comparison operations	3
Span aggregate operations	15
Property value activation	4
Constant translation	3
Time-stamp translation	3
Function binding	3
Calendric system activation	2

Table 2: Uniform Calendric Support Operations

Specifically, event and interval computations are calendar independent. Hence, operations on events and intervals, once translated into time-stamps, can be executed directly by the TADT. For example, event values, i.e., time-stamps in the physical representation, do not require a calendar interpretation; their time-stamps completely describe their values. Therefore, operations on event values, such as *event* `precedes` *interval*, are simple time-stamp manipulations that can be performed directly by the TADT. However, for operations involving span values, it is not known if the operation is calendar independent until the time-stamps of the operands are examined. Variable spans require calendar support while operations involving only fixed spans do not. The query processor is not capable of resolving this because the type system of the query language does not distinguish between variable spans and fixed spans. The TADT provides a routine `tadt_is_vs` which determines if a span is variable or fixed. The UCS calls this routine to determine if any operand is a variable span and, if so, invokes a calendar to perform the given operation. Otherwise, the operation is passed to the TADT which performs the computation. Almost two-thirds of the UCS routines are these simple "traffic-control" routines related to variable spans. See Section 9 for more details and examples of operations. We will describe in more detail the interface between the UCS and a calendar in Section 7.

6.1 External Data Structures

The UCS must maintain the function bindings specified by calls to `ucs_bind_function`. The structures and types used to manage this information follow.

```
typedef enum { arg_integer, arg_string, arg_event, arg_interval, arg_span } formal_arg_type;



typedef union {
          int integer;
```

```
          char *character;
          event_space_type event;
          interval_space_type interval;
          span_space_type span;
} *actual_arg_type;
```

The following data types are provided to accommodate both variable and fixed spans in aggregates. For variable spans, each calendar may define different aggregate states. The UCS aggregate states must be able to accomodate either fixed or variable spans. The aggregate states must be able to contain either any arbitrary contents (for variable spans) or some specific item (for fixed spans).

```
typedef enum { fixed, variable } span_types;



typedef struct {
          span_types tag;
          avg_state_fs fs_state;
          void *vs_state;
} *ucs_avg_state_s;



typedef struct {
          span_types tag;
          sum_state_fs fs_state;
          void *vs_state;
} *ucs_sum_state_s;



typedef struct {
          span_types tag;
          span_type fs_state;
          void *vs_state;
} *ucs_min_state_s;



typedef struct {
          span_types tag;
          span_type fs_state;
          void *vs_state;
} *ucs_max_state_s;
```

Finally, the UCS interacts with the individual calendars via a *value array*.

```
typedef int value_array_type[];
```

The following error codes are returned by the UCS when exceptional conditions are encountered.

```
typedef enum {
          ucs_ok,
          ucs_string_invalid, /* input event, interval or span invalid */
          ucs_incompatible_types, /* incompatible types on function bind */
          ucs_function_not_found, /* function not found on function bind */
          ucs_invalid_property, /* attempt to push an invalid calendar property */
          ucs_table_overflow, /* stack full on push */
          ucs_table_underflow, /* stack empty on pop */
} ucs_error_type;
```

6.2 Externally Visible Routines

We describe in this section the functions comprising the UCS.

6.2.1 External Data Structure Allocation

The following routines allocate externally visible data structures used by the UCS. The actual memory allocation is done by calling the TADT routine `tadt_malloc`. Many of the prologues are omitted here since they are very similar to the prologues in Section 5.2.1.

```
ucs_sum_state_s ucs_allocate_sum_state_s();

ucs_avg_state_s ucs_allocate_avg_state_s();

ucs_min_state_s ucs_allocate_min_state_s();

ucs_max_state_s ucs_allocate_max_state_s();
```

6.2.2 Span Arithmetic Support

The following are generic routines for span arithmetic operations. The run-time system of the query processor invokes these routines when performing arithmetic on either fixed or variable span time-stamps. Each procedure determines whether one of its operands is variable or not and either calls a TADT routine or a calendar routine as appropriate. A detailed example is given in Section 9.

```
/*
*Routine:  ucs_neg_s
*
*Description:  Return a negative span whose duration is the
*              same as the input span.
*
*Arguments:  spanin -- (IN) : span to be converted
*            result-- (OUT): result negative span
*
*Return Value:  Error code
*
*Errors:  semantic_error
*
*Side Effects:  Overwrites result
*/
error_type ucs_neg_s(span_type spanin, span_type result);



/*
*Routine:  ucs_s_add_s
*
*Description:  Compute the sum of two spans
*
*Arguments:  span1, span2 -- (IN) : spans to be added
*            result -- (OUT): the sum of the arguments
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type ucs_s_add_s(span_type span1, span_type span2, span_type result);



/*
*Routine:  ucs_s_minus_s
*
*Description:  Compute the difference of two spans
```

```
*
*Arguments:  span1 -- (IN) : minuend (span to be subtracted from)
*            span2 -- (IN) : subtrahend
*            result-- (OUT): the result of "span1 - span2"
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type ucs_s_minus_s(span_type span1, span_type span2, span_type result);



/*
*Routine:  ucs_e_add_s
*
*Description:  Displace an event by adding a span
*
*Arguments:  event_in -- (IN) : the event time
*            span -- (IN) : the increment period
*            result -- (OUT) : the new event time after displacement
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type ucs_e_add_s(event_type event_in, span_type span, event_type result);



/*
*Routine:  ucs_e_minus_s
*
*Description:  Displace an event by subtracting a duration of time (span)
*
*Arguments:  event_in -- (IN) : the event time
*            span -- (IN) : the decrement period
*            result -- (OUT) : the new event time after decrement
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type ucs_e_minus_s(event_type event_in, span_type span, event_type result);



/*
*Routine:  ucs_s_times_n
*
*Description:  This function multiplies a span by a numeric value
*
*Arguments:  span_in -- (IN) : the input span
*            n -- (IN) : number of times
*            result -- (OUT) : the result of (span x n)
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type ucs_s_times_n(span_type span_in, double n, span_type result);
```

```
/*
*Routine:  ucs_s_div_n
*
*Description:  Return a span equal to the division of the input span by
*              a numeric value
*
*Arguments:  span_in -- (IN) :the input span
*            n -- (IN) : number of division
*            result -- (OUT): the result of (span / n)
*
*Return Value:  Error code
*
*Errors:  divide_by_zero
*
*Side Effects:  Overwrites result
*/
error_type ucs_s_div_n(span_type span_in, double n, span_type result);


/*
*Routine:  ucs_s_div_s
*
*Description:  Divide one span by another
*
*Arguments:  span1 -- (IN): the dividend
*            span2 -- (IN): the divisor
*            dividend -- (OUT): the result of (s/s)
*
*Return Value:  Error code
*
*Errors:  divide_by_zero
*
*Side Effects:  Overwrites dividend
*/
error_type ucs_s_div_s(span_type span1, span_type span2, double *dividend);


/*
*Routine:  ucs_i_add_s
*
*Description:  Displace an interval by adding a duration (span) of time
*
*Arguments:  interval_in -- (IN) : the interval to be displaced
*            span -- (IN) : the span of displacement
*            result -- (OUT): interval displace by span
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type ucs_i_add_s (interval_type interval_in, span_type span, interval_type result);


/*
*Routine:  ucs_i_minus_s
*
*Description:  Displace an interval by subtracting a duration (span) of time
*
*Arguments:  interval_in -- (IN) : the interval to be displaced
*            span -- (IN) : the span of displacement
*            result -- (OUT) : interval displaced by span
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type ucs_i_minus_s (interval_type interval_in, span_type span, interval_type result);
```

6.2.3 Span Comparison Support

The following are generic routines for span comparison operations. The run-time system of the query processor invokes these routines when performing comparison on either fixed or variable span values. Each procedure determines whether one of its operands is variable or not and either calls a TADT routine or a calendar routine as appropriate. If a span is variable, then its calendar is called to perform the comparison.

```
/*
*Routine:  ucs_s_eq_s
*
*Description:  This routine compares two span values.  It returns
*              the boolean value TRUE if they are equal and
*              FALSE otherwise.
*
*Arguments:  span1, span2 -- (IN) : spans to be compared
*
*Return Value:  bool :  TRUE if span1 = span2
*               FALSE if span1 <> span2
*
*Errors:  None
*
*Side Effects:  None
*/
bool ucs_s_eq_s(span_type span1, span_type span2);



/*
*Routine:  ucs_s_lt_s
*
*Description:  This routine compares two span values.  It returns
*              the boolean value TRUE if the first span is less than
*              the second span and FALSE otherwise.
*
*Arguments:  span1, span2 -- (IN) : spans to be compared
*
*Return Value:  bool :  TRUE if span1 < span2
*               FALSE if span1 >= span2
*
*Errors:  None
*
*Side Effects:  None
*/
bool ucs_s_lt_s(span_type span1, span_type span2);



/*
*Routine:  ucs_s_gt_s
*
*Description:  This routine compares two span values.  It returns
*              the boolean value TRUE if the first span is greater than
*              the second span and FALSE otherwise.
*
*Arguments:  span1, span2 -- (IN) : spans to be compared
*
*Return Value:  bool :  TRUE if span1 > span2
*               FALSE if span1 <= span2
*
*Errors:  None
*
*Side Effects:  None
*/
bool ucs_s_gt_s(span_type span1, span_type span2);
```

6.2.4 Span Aggregate Function Support

The following are generic routines for span aggregate function operations. The run-time system of the query processor invokes these routines when performing aggregate functions operations on variable or fixed span values. Each procedure determines whether one of its operands is variable or not and either calls a TADT routine or a calendar routine as appropriate. Aggregate computations on variable span values are handled by a calendar. Aggregates involving both fixed and variable spans are handled by the TADT with calendar support.

We do not show the procedure prologues for these routines since they are very similar to those for the TADT aggregate routines.

```
int *ucs_init_count_s();
error_type ucs_accum_count_s(int *count_state, span_type span);
error_type ucs_final_count_s(int final_state, int *final_count);

ucs_sum_state_s ucs_init_sum_s();
error_type ucs_accum_sum_s(ucs_sum_state_s sum_state, span_type span);
error_type ucs_final_sum_s(ucs_sum_state_s final_s, span_type final_span_sum);

ucs_avg_state_s ucs_init_avg_s();
error_type ucs_accum_avg_s(ucs_avg_state_s avg_state, span_type span);
error_type ucs_final_avg_s(ucs_avg_state_s final_state, span_type final_avg_span);

ucs_max_state_s ucs_init_max_s();
error_type ucs_accum_max_s(ucs_max_state_s max_state, span_type span);
error_type ucs_final_max_s(ucs_max_state_s final_state, span_type final_max_span);

ucs_min_state_s ucs_init_min_s();
error_type ucs_accum_min_s(ucs_min_state_s min_state, span_type span);
error_type ucs_final_min_s(ucs_min_state_s final_state, span_type final_min_span);
```

6.2.5 Property Management

The use of properties adds flexibility to the calendars, and it helps tailoring the calendars to the needs of different users. The properties may be used to specify the input and output formats of temporal constants, the current locale, an override input epoch, and the naming of the concepts beginning and forever. The current values of the properties in effect are maintained by the UCS.

The properties and their uses follow.

- *Locale*—the curent location, e.g., the city where the computer user is located. This quantity may be used for time zone displacement.
- *Event Input Format*—the format of event temporal constants that are input to the database. A value of this property is a meta-string which is described elsewhere [Soo & Snodgrass 1992A]. It is used by the uniform calendric system and a calendar to parse an input string. The values of the following five properties are also meta-strings.
- *Event Output Format*—the format to be used when converting events to strings.
- *Span Input Format*—the format of span temporal constants that are input to the database.
- *Span Output Format*—the format for output of spans.
- *Interval Input Format*—the format for input of intervals.
- *Interval Output Format*—the output format of intervals.

- *Override Input Epoch*—the epoch that is consulted the first when transforming input strings into time-stamps within the current calendric system. It overrides the default order for the calendric system.
- *Beginning*—the name for a special event value preceding any other. This special event value is similar to $-\infty$. This property is provided so that language specific names may be used.
- *Forever*—the name for a special event value following any other. This event value is similar to $+\infty$.

The *Locale* property is calendar specific, and its values are interpreted by the calendar. The other properties are not calendar specific. Table 3 illustrates a sample default property table; this table is incomplete, but conveys the general notion of what the various properties express. An example of property value activation is discussed in Section 9.4. Section 9.5 has a more complete discussion of the use of the input and output formats.

Property Name	*Property Value*
Locale	`Chicago`
Event Input Format	*Month/Day/Year*
Event Output Format	*Month/Day/Year*
Span Input Format	*Event - Event*
Span Output Format	*Event - Event*
Interval Input Format	*Event*
Interval Output Format	*Event*
Override Input Epoch	`Gregorian`
Beginning	`da beginning`
Forever	`da end`

Table 3: Example Initial Property List

UCS support for property value activation is provided by the following routines. The run-time system of the query processor invokes these routines to activate and deactivate values of the properties.

```
/*
*Routine:  ucs_property_push
*
*Description:  Add a property to be activated
*
*Arguments:  property -- (IN): a property
*            valud -- (IN): the value of that property
*
*Return Value:  Error code
*
*Errors:  ucs_invalid_property
*         ucs_table_overflow
*
*Side Effects:  Adds the property to the list of properties
*               waiting to be activated
*/
ucs_error_type ucs_property_push(char *property, char *value);



/*
*Routine:  ucs_property_activate
*
*Description:  Activate any waiting properties
```

```
*
*Arguments:  None
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Activates any pushed properties
*/
ucs_error_type ucs_property_activate();


/*
*Routine:  ucs_property_clear
*
*Description:  Discard any properties waiting to be activated
*
*Arguments:  None
*
*Return Value:  Error code
*
*Errors:  None
*
*Side Effects:  Discards any waiting properties
*/
ucs_error_type ucs_property_clear();


/*
*Routine:  ucs_property_deactivate
*
*Description:  Deactivates the last set of activated properties
*
*Arguments:  None
*
*Return Value:  Error code
*
*Errors:  ucs_table_underflow
*
*Side Effects:  Uncovers the previously active set of properties
*/
ucs_error_type ucs_property_deactivate();
```

6.2.6 Temporal Constant Translation

The string to time-stamp conversion functions of the UCS are as follows. The query processor invokes these routines at run-time when converting temporal constants to time-stamps. Each of these conversion routines parses its input string according to the property table specification. Examples of how these routines are used are provided in Section 9.

```
/*
*Routine:  ucs_e_string_to_e
*
*Description:  Convert an event string to a timestamp according to the
*              specification of the Event Input Format, Locale,
*              Override Input Epoch, Beginning, and Forever properties
*
*Arguments:  string -- (IN) : the event in calendar specific form
*            event -- (OUT): timestamp for the event
*
*Return Value:  Error code
*
*Errors:  ucs_string_invalid
*
*Side Effects:  Overwrites event
*/
ucs_error_type ucs_e_string_to_e(char *string, event_type event);
```

```
/*
*Routine:  ucs_i_string_to_i
*
*Description:  Convert an interval string to a timestamp according to the
*              specification of the Interval Input Format, Locale, and
*              Override Input Epoch properties
*
*Arguments:  string -- (IN) : the interval in calendar specific form
*            interval -- (OUT): timestamp for the interval
*
*Return Value:  Error code
*
*Errors:  ucs_string_invalid
*
*Side Effects:  Overwrites interval
*/
ucs_error_type ucs_i_string_to_i(char *string, interval_type interval);



/*
*Routine:  ucs_s_string_to_s
*
*Description:  Convert a span string to a timestamp according to the
*              specification of the Span Input Format and Override
*              Input Epoch properties
*
*Arguments:  string -- (IN) : the span in calendar specific form
*            span -- (OUT): timestamp for the span
*
*Return Value:  Error code
*
*Errors:  ucs_string_invalid
*
*Side Effects:  Overwrites span
*/
ucs_error_type ucs_s_string_to_s(char *string, span_type span);
```

6.2.7 Time-stamp Translation Support

The time-stamp to string conversion functions of the UCS are as follows. The query processor invokes these routines at run-time when converting time-stamps to strings for output to procedure parameters. This interaction between the UCS and calendars is described in detail in Section 9.

```
/*
*Routine:  ucs_e_to_string
*
*Description:  Convert a timestamp to an event string according to the
*              the specification of the Event Output Format,
*              Locale, Beginning, and Forever properties
*
*Arguments:  event -- (IN) : the timestamp to be converted
*            str -- (OUT): the string to be returned
*
*Return Value:  Error code
*
*Errors:  conversion_error
*         string_overflow
*
*Side Effects:  Overwrites str
*/
error_type ucs_e_to_string(event_type event, string_struct str);



/*
*Routine:  ucs_i_to_string
*
```

```
*Description:  Convert a timestamp to an interval string according to the
*              specification of the Interval Output Format and
*              Locale properties
*
*Arguments:  interval -- (IN) : the timestamp to be converted
*            str -- (OUT): the string to be returned
*
*Return Value:  Error code
*
*Errors:  conversion_error
*         string_overflow
*
*Side Effects:  Overwrites str
*/
error_type ucs_i_to_string(interval_type interval, string_struct str);



/*
*Routine:  ucs_s_to_string
*
*Description:  Convert a timestamp to a span string according to the
*              specification of the Span Output Format property
*
*Arguments:  span -- (IN) : the timestamp to be converted
*            str -- (OUT): the string to be returned
*
*Return Value:  Error code
*
*Errors:  conversion_error
*         string_overflow
*
*Side Effects:  Overwrites str
*/
error_type ucs_s_to_string(span_type span, string_struct str);
```

6.2.8 Calendar Function Binding and Invocation

Each calendar specific function uses a special format to describe the parameters that it takes and the return type. These parameters are passed to the function `ucs_bind_function`, which performs name resolution and type checking on the formal parameters, and returns a function pointer. The function parameters are described by passing a pointer to a list of enumerated type values describing the actual function parameters. The size of the list is also passed. The return type, a similar type description value, is also computed by the `ucs_bind_function`.

The function pointer that is returned may be called directly. Each of these functions are called with two parameters: the address of a list of pointers to the actual parameters, and a pointer to a return structure of the appropriate type. This return structure has been pre-allocated before calling this function. If the return value varies in size, e.g., an `event_type`, then the maximum allowable size structure is allocated. The actual return value is an error code indicating the success or failure of the function. If any coercion of values needs to be done, then it is done before calling the bound function. All the data structures passed to the bound function should be de-allocated after the function returns.

The semantic analyzer of the query processing system may determine if a function exists by calling the routine `ucs_function_exists`. To actually bind a function, use `ucs_bind_function`.

```
/*
*Routine:  ucs_bind_function
*
*Description:  Resolve a calendric defined function name
*
*Arguments:  function_name -- (IN) : name of the function
```

```
*            num_arguments -- (IN) : number of arguments in the array
*            type_array -- (IN) : array of function arguments codes
*            return_type_type -- (OUT): return value type for function
*            f -- (OUT): handle to the function
*
*Return Value:  Error code
*
*Errors:  ucs_incompatible_types
*         ucs_function_not_found
*
*Side Effects:  Overwrites f
*/
ucs_error_type ucs_bind_function(char *function_name, int num_arguments, formal_arg_type type_array[],
formal_arg_type *return_type_type, generic_function_handle f);



/*
*Routine:  ucs_function_exists
*
*Description:  Determines if a function exists
*
*Arguments:  function_name -- (IN) : name of the function
*
*Return Value:  bool -- TRUE if the function exists
*               FALSE if the function does not exist
*
*Errors:  None
*
*Side Effects:  None
*/
bool ucs_function_exists(char *function_name);



/*
*Routine:  generic_function_handle
*
*Description:  The generic description of the function pointer returned by
*              ucs_bind_function
*
*Arguments:  array -- (IN) : array of parameter pointers
*            return_val -- (OUT) : pointer to the return value
*
*Return Value:  semantic_error
*
*Errors:  Depends on the function
*
*Side Effects:  Overwrites return_val
*/
typedef error_type (generic_function_handle)(actual_arg_type array[], actual_arg_type *return_val);
```

Definitions of the parameter and return types for the function binding procedure are in Section 6.1.

To invoke a function, just call it with the correct parameters. At function binding time (compile time), the UCS must determine which calendar has defined this function. If the function name is unique, then of course there is no ambiguity. Otherwise, the proper calendar function may be determined by the lexical scoping of the calendric system.

6.2.9 Calendric System Activation Support

The calendric system activation functions of the UCS are shown below. These functions are used by the query processor during semantic analysis to check that calendric systems exist, and to activate calendric systems during both semantic analysis (for calendar defined function binding) and run-time (for temporal constant interpretation).

```
/*
*Routine:  ucs_is_calendric_system
*
*Description:  Determines if the parameter is the name of a calendric system
*
*Arguments:  calendric_system_name -- (IN) : the name of the calendric system
*
*Return Value:  bool :  TRUE if the calendric system is defined
*               FALSE otherwise
*
*Errors:  None
*
*Side Effects:  None
*/
bool ucs_is_calendric_system(char *calendric_system_name);



/*
*Routine:  ucs_declare_calendric_system
*
*Description:  Inform uniform calendric support that a
*              new calendric system has been selected
*
*Arguments:  calendric_system_name -- (IN) : the name of the calendric system
*
*Return Value:  Error code
*
*Errors:  None, assumes that ucs_is_calendric_system was called previously
*
*Side Effects:  Changes current calendric system in UCS
*/
error_type ucs_declare_calendric_system(char *calendric_system_name);
```

6.3 Internal Data Structures

As previously mentioned, a calendric system is represented entirely by data structures within the UCS; calendric systems contain no procedural components. From an architectural standpoint, a calendric system exists solely to integrate calendars and to supply a mechanism for accessing the facilities those calendars provide. As such, static data structures identifying calendars and the services exported by those calendars are all that is needed to implement a calendric system.

A calendric system data structure contains three components, the name of the calendric system, the calendars and epochs defined for the calendric system, and a list of routines. Calendric systems and their constituent calendars were discussed in Section 2. The set of routines defined via the calendric system is the collection of routines defined by each calendar of the calendric system. This list is collected by the DBMS generation toolkit described in Section 10.

To illustrate how a calendric system is represented, we return to the Russian calendric system of Section 2. The Russian calendric system is composed of six calendars, the `geologic`, `carbon-14`, `roman`, `julian`, `communist`, and `gregorian` calendars [Soo & Snodgrass 1992B]. Each calendar is associated with one epoch, except for the `gregorian` calendar which has two epochs. The `num_epochs` field in the `calendric_system` structure is thus set to `7`, and 7 `epoch` structures are allocated. As illustrated by the Russian calendric system, a single calendar may be in effect during several epochs—the only requirement is that no epochs in a calendric system overlap. The actual calendar structures are generated by the generation toolkit (Section 10).

When temporal constants are encountered in a query, a calendar of the calendric system must be selected to translate the constant into a time-stamp. The calendars must be prioritized for this purpose. For our example, it is reasonable to assume that the largest number of temporal constants encountered will belong to the present epoch (covered by the `gregorian` calendar), followed by the

two year epoch of the communist calendar, followed by the epoch from 1917 to 1929 (also covered by the gregorian calendar), followed by the epoch prior to the revolution (covered by the julian calendar), and lastly the epochs covered by the roman carbon-14, and geologic calendars in the indicated order. Therefore, the DBI would assign the highest input priorities to the gregorian calendar, the next highest priority to the julian calendar, etc. Then, when a temporal constant is evaluated, the UCS will, after preprocessing the input, pass the constant to each calendar according to the priorities. The time-stamp returned by the first calendar to successfully translate the constant is taken as the value of the constant.

Each calendar may define functions invokable by the query language. These functions are bound at compile time by the function ucs_bind_function. This was discussed in Section 6.2.8.

In summary, each calendric system is composed of a name and several epochs. The epochs are non-overlapping and have an associated calendar. The following structures are used to specify these epochs. Note that the calendars are listed in decreasing input priority with the epoch at the head of the list having the highest input priority.

```
typedef struct {
          interval_type epoch;
          calendar_type *calendar;
} epoch_type;



typedef struct {
          char *name;
          int num_epochs;
          epoch_type epochs[];
} calendric_system_type;



calendric_system_type calendric_system_list[]; /* list of all calendric systems */
```

The following structures are constructed for particular calendars by the generation toolkit, and are used by the UCS.

```
typedef struct {
          char *name;
          function_entry_type functions[];
          int field_index_table_size;
          char *field_index_table[];
} calendar_type;



typedef struct {
          char *function_name;
          int num_params;
          formal_arg_type list_params[];
          formal_arg_type return_type;
} function_entry_type;
```

Property values are managed by the UCS. Internally, property values are not represented by strings, rather a more efficient method is used. Two routines are needed in order to update the property table. First the query processor inserts individual values of each property to be updated into the property table, using the routine ucs_property_push. If more than one value of the same property is inserted, only the most recent value is retained. Second, all the previously inserted properties are activalted as a group, using the routine ucs_property_activate. The routine

ucs_property_deactivate reverses the effect of the most recent ucs_property_activate, and makes the previous property table the current property table. When the initial property table is in effect, calling this routine results in an error—the initial property table cannot be undone.

Property management is implemented in a very simple manner. Property values are recorded in a stack, and a table of pointers, one for each property, points to the current value of each of the ten properties. The stack maintains the history of property activation, and the table provides efficient access to the current property values. Initially, the stack contains only the default properties, and the pointers point to the default property values in the stack. During execution, a call to ucs_property_push will push a new property value onto the stack. If this property value is then activated using ucs_property_activate, the pointer associated with that property in the table of pointers is updated to point to the most recently activated value.

For efficiency, each property in the stack points back to its previous value; when a ucs_property_deactivate is performed and it is necessary to find the old value of that property, the stack does not need to be searched.

```
typedef struct {
          char *value;
          struct property_element *previous_value;
} *property_element_type;

#define NUM_PROPERTIES 10

property_element_type initial_property_list[NUM_PROPERTIES];
```

7 Calendars

The calendar is the most critical component of the architecture. It represents the local adaptation of temporal semantics within the architecture, and so the majority of its contents must be provided by the DBI. These contents include calendar unique functions, routines supporting temporal constant evaluation and time-stamp evaluation, and calendar dependent aggregate, arithmetic, and comparison operations. These routines constitute the services exported by the calendar to the UCS.

Table 4 identifies the operations that must be programmed as part of a calendar implementation. We note that most of these operations involve variable spans. If the calendar does not define any variable spans, then these functions are not required, and only the six remaining routines are required to define a calendar. If only one variable span is defined, then the variable span routines can be quite simple. This is the case with our initial description of the Gregorian calendar which has a single variable span, month. When multiple variable spans are defined, then each routine must contend with all, and some must handle the more complex combinations of two variable spans.

Constructing calendar routines may be difficult for the DBI. Consequently, whenever possible we have identified common processing that must be present in all calendars and shifted that code into the UCS and TADT to minimize the DBI's programming effort. Shifting processing to the UCS is made possible by using table-driven algorithms. Calendars provide field index tables to the UCS; the UCS uses this information to interpret input data or construct output data. In particular, properties allow local adaptation of calendar semantics. At the query language level, properties are used to parameterize calendars, and property values affect the result of calendar operations. However, at the architectural level, our goal is to simplify the implementation of calendars as much as possible. Consequently, we have moved the interpretation and application of property values out

of the calendar and into the UCS. Calendars are not required to interpret property values directly, and whenever possible, the UCS pre-processes the data to apply the effects of property values.

Operation	*Number of Routines*
Time-stamp translation	3
Constant translation	3
Memory allocation for span states	4
Variable span arithmetic operations	15
Variable span comparison operations	8
Variable span aggregate operations	15
Variable span to fixed span conversion	1
Calendar specific functions	*calendar dependent*

Table 4: Calendar Operations

7.1 External Data Structures

These structures are allocated indirectly by the TADT via the `cal_`*calendar*`_..._fstate_to_vstate` routines. They are arbitrarily-sized data structures defined by each calendar.

```
typedef void *count_state_vs;
typedef void *sum_state_vs;
typedef void *avg_state_vs;
typedef void *min_state_vs;
typedef void *max_state_vs;
```

7.2 Externally Visible Routines

We describe in this section the functions comprising a calendar definition. For each different calendar system, the word *calendar* in each function name would be replaced by the name of the actual calendar. For example, in the Gregorian calendric system, there would be a function called `cal_gregorian_gen_s_array`.

7.2.1 Data Structure Allocation

The following routines are used by the calendar module to allocate memory for data structures that are externally visible. The actual memory allocation is done by calling the TADT routine `tadt_malloc`.

```
sum_state_vs cal_allocate_sum_state_vs();

avg_state_vs cal_allocate_avg_state_vs();

min_state_vs cal_allocate_min_state_vs();

max_state_vs cal_allocate_max_state_vs();
```

7.2.2 Time-stamp Translation

The following routines support translation of time-stamps into strings (temporal constants). They are required of all calendars.

```
/*
*Routine:  cal_calendar_gen_s_array
*
*Description:  Create a parsed version of the time-stamp
*
*Arguments:  span -- (IN) : time-stamp
*            val_array -- (OUT): parsed version of time-stamp
*
*Return Value:  Error code
*
*Errors:  conversion_error
*
*Side Effects:  Overwrites val_array
*/
error_type cal_calendar_gen_s_array(span_type span,value_array_type val_array);



/*
*Routine:  cal_calendar_gen_e_array
*
*Description:  Create a parsed version of the time-stamp
*
*Arguments:  event -- (IN) : time-stamp
*            val_array -- (OUT): parsed version of time-stamp
*
*Return Value:  Error code
*
*Errors:  conversion_error
*
*Side Effects:  Overwrites val_array
*/
error_type cal_calendar_gen_e_array(event_type event, value_array_type val_array);



/*
*Routine:  cal_calendar_gen_i_array
*
*Description:  Create a parsed version of the time-stamp
*
*Arguments:  event -- (IN) : time-stamp
*            val_array_1 -- (OUT): first half of parsed version of time-stamp
*            val_array_2 -- (OUT): second half of parsed version of time-stamp
*
*Return Value:  Error code
*
*Errors:  conversion_error
*
*Side Effects:  Overwrites val_array_1 and val_array_2
*/
error_type cal_calendar_gen_i_array(event_type event, value_array_type val_array_1,
                                    value_array_type val_array_2);
```

7.2.3 Constant Translation

The following routines support translation of strings (temporal constants) into time-stamps. They are required of all calendars.

```
/*
*Routine:  cal_calendar_gen_s_timestamp
*
*Description:  Create a span time-stamp from a parsed version of the
*              input string
*
*Arguments:  val_array -- (IN) : parsed representation of the input string
*            span -- (OUT): span time-stamp
*
```

```
*Return Value:  Error code
*
*Errors:  conversion_error
*
*Side Effects:  Overwrites span
*/
error_type cal_calendar_gen_s_timestamp(value_array_type val_array, span_type span);



/*
*Routine:  cal_calendar_gen_e_timestamp
*
*Description:  Create an event time-stamp from a parsed version of the
*              input string
*
*Arguments:  val_array -- (IN) : parsed representation of the input string
*            locale -- (IN) : current locale property value
*            event -- (OUT): event time-stamp
*
*Return Value:  Error code
*
*Errors:  conversion_error
*
*Side Effects:  Overwrites event
*/
error_type cal_calendar_gen_e_timestamp(value_array_type val_array, char *locale, event_type event);



/*
*Routine:  cal_calendar_gen_i_timestamp
*
*Description:  Create an interval time-stamp from a parsed version of the
*              input string
*
*Arguments:  val_array1 -- (IN) : parsed representation of beginning of interval
*            val_array2 -- (IN) : parsed representation of ending of interval
*            locale -- (IN) : current locale property value
*            interval -- (OUT): interval time-stamp
*
*Return Value:  Error code
*
*Errors:  conversion_error
*
*Side Effects:  Overwrites interval
*/
error_type cal_calendar_gen_i_timestamp(value_array_type val_array1, value_array_type val_array2,
                                        char *locale, interval_type interval);
```

7.2.4 Span Arithmetic Support

The following variable span arithmetic functions are required if the calendar uses any variable spans.

```
/*
*Routine:  cal_calendar_neg_s
*
*Description:  Return a negative span whose duration is the
*              same as the input span
*
*Arguments:  spanin -- (IN) : span to be converted
*            result --(OUT): result negative span
*
*Return Value:  Error code
*
*Errors:  None
*
```

```
*Side Effects:  Overwrites result
*/
error_type cal_calendar_neg_s(span_type spanin, span_type result);



/*
*Routine:  cal_calendar_vs_add_fs
*
*Description:  Compute the sum of a variable span and a fixed span
*
*Arguments:  vs -- (IN) : variable span
*            fs -- (IN) : fixed span
*            result-- (OUT): the sum of the arguments
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_vs_add_fs(span_type vs, span_type fs, span_type result);



/*
*Routine:  cal_calendar_vs_add_vs
*
*Description:  Compute the sum of two variable spans
*
*Arguments:  vs1, vs2 -- (IN) : spans to be added
*            result -- (OUT): result of sum of two spans
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_vs_add_vs(span_type vs1, span_type vs2, span_type result);



/*
*Routine:  cal_calendar_vs_minus_fs
*
*Description:  Subtract a fixed span from a variable span
*
*Arguments:  vs -- (IN) : variable span
*            fs -- (IN) : fixed span
*            result-- (OUT): the difference of the arguments
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_vs_minus_fs(span_type vs, span_type fs, span_type result);



/*
*Routine:  cal_calendar_fs_minus_vs
*
*Description:  Subtract a fixed span from a variable span
*
*Arguments:  fs -- (IN) : fixed span
*            vs -- (IN) : variable span
*            result-- (OUT): the difference of the arguments
*
```

```
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_fs_minus_vs(span_type fs, span_type vs, span_type result);



/*
*Routine:  cal_calendar_vs_minus_vs
*
*Description:  Subtract a variable span from a variable span
*
*Arguments:  vs1 -- (IN) : variable span
*            vs2 -- (IN) : variable span
*            result-- (OUT): the difference of the arguments
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_vs_minus_vs(span_type vs1, span_type vs2, span_type result);



/*
*Routine:  cal_calendar_vs_add_e
*
*Description:  Add a variable span to an event
*
*Arguments:  vs -- (IN) : variable span
*            event_in -- (IN) : event
*            result -- (OUT): the sum of the arguments
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_vs_add_e(span_type vs, event_type event_in, event_type result);



/*
*Routine:  cal_calendar_e_minus_vs
*
*Description:  Subtract a variable span from an event
*
*Arguments:  event_in -- (IN) : event
*            vs -- (IN) : variable span
*            result -- (OUT): the difference of the arguments
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_e_minus_vs(event_type event_in, span_type vs, event_type result);



/*
*Routine:  cal_calendar_vs_times_n
*
*Description:  Multiply a variable span by a numeric value
*
*Arguments:  vs -- (IN) : variable span
```

```
*            n -- (IN) : numeric value
*            result-- (OUT): the product of the arguments
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_vs_times_n(span_type vs, double n, span_type result);



/*
*Routine:  cal_calendar_vs_div_n
*
*Description:  Divide a variable span by a numeric value
*
*Arguments:  vs -- (IN) : variable span
*            n -- (IN) : numeric value
*            result -- (OUT): the division of vs by n
*
*Return Value:  Error code
*
*Errors:  None (assumes that UCS checks that n is not 0)
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_vs_div_n(span_type vs, double n, span_type result);



/*
*Routine:  cal_calendar_fs_div_vs
*
*Description:  Divide a fixed span by a variable span
*
*Arguments:  fs -- (IN) : fixed span dividend
*            vs -- (IN) : variable span divisor
*            dividend -- (OUT): the division of fs by vs
*
*Return Value:  Error code
*
*Errors:  divide_by_zero
*
*Side Effects:  Overwrites dividend
*/
error_type cal_calendar_fs_div_vs(span_type fs, span_type vs, double *dividend);



/*
*Routine:  cal_calendar_vs_div_fs
*
*Description:  Divide a variable span by a fixed span
*
*Arguments:  vs -- (IN) : variable span dividend
*            fs -- (IN) : fixed span divisor
*            dividend -- (OUT): the division of vs by fs
*
*Return Value:  Error code
*
*Errors:  None (assume that UCS checks to make sure fs is not 0)
*
*Side Effects:  Overwrites dividend
*/
error_type cal_calendar_vs_div_fs(span_type vs, span_type fs, double *dividend);
```

```
/*
*Routine:  cal_calendar_vs_div_vs
*
*Description:  Divide a variable span by a variable span
*
*Arguments:  vs1 -- (IN) : variable span dividend
*            vs2 -- (IN) : variable span divisor
*            dividend -- (OUT): the division of vs1 by vs2
*
*Return Value:  Error code
*
*Errors:  semantic_error
*
*Side Effects:  Overwrites dividend
*/
error_type cal_calendar_vs_div_vs(span_type vs1, span_type vs2, double *dividend);



/*
*Routine:  cal_calendar_i_add_vs
*
*Description:  Add a variable span to an interval
*
*Arguments:  interval_in -- (IN) : interval
*            vs -- (IN) : variable span
*            result -- (OUT): the sum of the arguments
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_vs_add_i(span_type vs, interval_type interval_in, interval_type result);



/*
*Routine:  cal_calendar_i_minus_vs
*
*Description:  Subtract a variable span from an interval
*
*Arguments:  interval_in -- (IN) : the interval to be displaced
*            vs -- (IN) : the span of displacement
*            result -- (OUT): the difference of the arguments
*
*Return Value:  Error code
*
*Errors:  out_of_range
*
*Side Effects:  Overwrites result
*/
error_type cal_calendar_i_minus_vs(interval_type interval_in, span_type vs, interval_type result);
```

7.2.5 Span Comparison Support

The following variable span comparison routines are required if the calendar uses any variable spans.

```
/*
*Routine:  cal_calendar_vs_eq_fs
*
*Description:  Compare a variable span to a fixed span.  Returns
*              the boolean value TRUE if they are equal and
*              FALSE otherwise.
*
*Arguments:  vs -- (IN) : variable span
```

```
*            fs -- (IN) : fixed span
*
*Return Value:  bool :  TRUE if vs = fs
*               FALSE if vs <> fs
*
*Errors:  semantic_error
*
*Side Effects:  None
*/
bool cal_calendar_vs_eq_fs(span_type vs, span_type fs);



/*
*Routine:  cal_calendar_vs_eq_vs
*
*Description:  Compare a variable span to a variable span.  Returns
*              the boolean value TRUE if they are equal and
*              FALSE otherwise.
*
*Arguments:  vs1 -- (IN) : variable span
*            vs2 -- (IN) : variable span
*
*Return Value:  bool :  TRUE if vs1 = vs2
*               FALSE if vs1 <> vs2
*
*Errors:  semantic_error
*
*Side Effects:  None
*/
bool cal_calendar_vs_eq_vs(span_type vs1, span_type vs2);



/*
*Routine:  cal_calendar_vs_lt_fs
*
*Description:  Compare a variable span to a fixed span.  Returns
*              the boolean value TRUE if the first span is less than
*              the second span and FALSE otherwise.
*
*Arguments:  vs -- (IN) : variable span
*            fs -- (IN) : fixed span
*
*Return Value:  bool:  TRUE if vs < fs
*               FALSE if vs >= fs
*
*Errors:  semantic_error
*
*Side Effects:  None
*/
bool cal_calendar_vs_lt_fs(span_type vs, span_type fs);



/*
*Routine:  cal_calendar_fs_lt_vs
*
*Description:  Compare a fixed span to a variable span.  Returns
*              the boolean value TRUE if the first span is less than
*              the second span and FALSE otherwise.
*
*Arguments:  fs -- (IN) : fixed span
*            vs -- (IN) : variable span
*
*Return Value:  bool :  TRUE if fs < vs
*               FALSE if fs >= vs
*
*Errors:  semantic_error
*
*Side Effects:  None
*/
bool cal_calendar_fs_lt_vs(span_type fs, span_type vs);
```

```
/*
*Routine:  cal_calendar_vs_lt_vs
*
*Description:  Compare a variable span to a variable span.  Returns
*              the boolean value TRUE if the first span is less than
*              the second span and FALSE otherwise.
*
*Arguments:  vs1 -- (IN) : variable span
*            vs2 -- (IN) : variable span
*
*Return Value:  bool :  TRUE if vs1 < vs2
*               FALSE if vs1 >= vs2
*
*Errors:  semantic_error
*
*Side Effects:  None
*/
bool cal_calendar_vs_lt_vs(span_type vs1, span_type vs2);



/*
*Routine:  cal_calendar_fs_gt_vs
*
*Description:  Compare a fixed span to a variable span.  Returns
*              the boolean value TRUE if the first span is greater than
*              the second span and FALSE otherwise.
*
*Arguments:  fs -- (IN) : fixed span
*            vs -- (IN) : variable span
*
*Return Value:  bool:  TRUE if fs > vs
*               FALSE if fs <= vs
*
*Errors:  semantic_error
*
*Side Effects:  None
*/
bool cal_calendar_fs_gt_vs(span_type fs, span_type vs);



/*
*Routine:  cal_calendar_vs_gt_fs
*
*Description:  Compare a variable span to a fixed span.  Returns
*              the boolean value TRUE if the first span is greater than
*              the second span and FALSE otherwise.
*Arguments:  vs -- (IN) : variable span
*            fs -- (IN) : fixed span
*
*Return Value:  bool :  TRUE if vs > fs
*               FALSE if vs <= fs
*
*Errors:  semantic_error
*
*Side Effects:  None
*
*/
bool cal_calendar_vs_gt_fs(span_type vs, span_type fs);



/*
*Routine:  cal_calendar_vs_gt_vs
*
*Description:  Compare a variable span to a variable span.  Returns
*              the boolean value TRUE if the first span is greater than
*              the second span and FALSE otherwise.
*
*Arguments:  vs1 -- (IN) : variable span
*            vs2 -- (IN) : variable span
```

```
*
*Return Value:  bool :  TRUE if vs1 < vs2
*               FALSE if vs1 >= vs2
*
*               Return Value:  Error code
*
*Errors:  semantic_error
*
*Side Effects:  None
*
*/
bool cal_calendar_vs_gt_vs(span_type vs1, span_type vs2);
```

7.2.6 Span Aggregate Function Support

The following variable span aggregate routines are required if the calendar uses any variable spans. In addition, the calendar must provide conversion routines for state conversion from fixed span to variable span aggregates. A detailed example of span aggregate computation is given in Section 9.3.

We show only one prologue for the aggregate routines. The remaining prologues are similar to those in the TADT.

```
/*
*Routine:  cal_calendar_count_fstate_to_vstate
*
*Description:  Convert a partial result of the count
*              of fixed spans to a count of variable spans
*
*Arguments:  fs_count_state -- (IN) : the count of spans so far
*            vs_count_state -- (OUT): the count of fixed spans
*
*Return Value:  Error code
*
*Errors:  conversion_error
*
*Side Effects:  Overwrites vs_count_state
*/
error_type cal_calendar_count_fstate_to_vstate(int *fs_count_state, void *vs_count_state);

error_type cal_calendar_accum_count_s(void *count_state, span_type span);
error_type cal_calendar_final_count_s(void *final_state, int *final_count);


error_type cal_calendar_sum_fstate_to_vstate(span_type fs_sum_state, void *vs_sum_state);
error_type cal_calendar_accum_sum_s(void *sum_state, span_type span);
error_type cal_calendar_final_sum_s(void *final_state, span_type span);


error_type cal_calendar_avg_fstate_to_vstate(avg_state_fs fs_avg_state, void *vs_avg_state);
error_type cal_calendar_accum_avg_s(void *avg_state, span_type span);
error_type cal_calendar_final_avg_s(void *final_state, span_type span);


error_type cal_calendar_max_fstate_to_vstate(span_type fs_max_state, void *vs_max_state);
error_type cal_calendar_accum_max_s(void *max_state, span_type span);
error_type cal_calendar_final_max_s(void *final_state, span_type span);


error_type cal_calendar_min_fstate_to_vstate(span_type fs_min_state, void *vs_min_state);
error_type cal_calendar_accum_min_s(void *min_state, span_type span);
error_type cal_calendar_final_min_s(void *final_state, span_type span);
```

7.2.7 Variable Span to Fixed Span Conversion

The following variable span to fixed span conversion routine is required if the calendar uses any variable spans.

```
/*
*Routine:  cal_calendar_vs_to_fs
*
*Description:  Converts a variable span to an equivalent fixed span
*
*Arguments:  vs -- (IN) : variable span
*            fs -- (OUT): fixed span
*
*Return Value:  Error code
*
*Errors:  semantic_error
*
*Side Effects:  Overwrites fs
*/
error_type cal_calendar_vs_to_fs(span_type vs, span_type fs);
```

7.3 Internal Data Structures

A calendar does not need internal data structures to store information. This is because the operation of a calendar is mostly stateless, with two exceptions. Overrides on default *Locale* property, as maintained in the UCS property stack, may affect the input and output of time-stamps. The current value of this property is passed into the appropriate conversion routines, as discussed in Sections 7.2.2 and 7.2.3. The other exception is the state of an aggregate computation, which is discussed in Sections 7.1 and 9.3.

8 Field Value Support

This section describes the module providing access to field value tables and field value routines. Field value support has been separated from calendars in the architecture because distinct calendars may make use of the same field value tables and routines.

8.1 External Data Structures

```
typedef enum {
          fv_ok, /*successful completion of the routine */
          fv_index_too_large, /*no string found, a too large index */
          fv_index_too_small, /*no string found, a too small index */
          fv_string_unrecognized, /*no index value found, unrecognized string */
          fv_string_overflow, /*string too long */
          fv_table_number_undefined, /*no table has this number */
          fv_table_name_undefined /*no table has this name */
} fv_error_type;
```

8.2 Externally Visible and Internal Routines

The field value support (FV) module exports the two routines described below. The first routine looks up an index value in an argument field value table and returns the corresponding string. Field value tables relate index values and text strings. Such tables may be implemented as either genuine tables or as routines. The second routine looks up a text string in an argument field value table and returns the corresponding index value.

```
/*
*Routine:  fv_index_to_string
*
*Description:  Return the string corresponding to an index value
*
*Arguments:  table_number -- (IN) : the argument table
*            index -- (IN) : the index value to convert to a string
*            result -- (OUT) : the requested string
*
*Return Value:  Error code
*
*Errors:  fv_index_too_large
*         fv_index_too_small
*         fv_table_number_undefined
*         fv_string_overflow
*
*Side Effects:  Assignment to passed result string
*/
fv_error_type fv_index_to_string(int table_number, int index, string_struct result);

/*
*Routine:  fv_string_to_index
*
*Description:  Return the index value of a string
*
*Arguments:  table_number -- (IN) : the argument table
*            string -- (IN) : the string to be located
*            result -- (OUT) : the index value of the string
*            num_char -- (OUT) : the number of recognized characters in the string
*
*Return Value:  Error code
*
*Errors:  fv_string_unrecognized
*         fv_table_number_undefined
*
*Side Effects:  Assignment to result and num_char
*/
fv_error_type fv_string_to_index(int table_number, char *string, int *result, int *num_char);
```

The next routine, also exported by the FV module, returns the number of a field value table when given the name of the table. It is used when converting the table names of format strings to their corresponding index values. Recall that a field value table may be implemented by a pair of functions. Only one function in a pair of functions is needed at a time; the routine requesting a table lookup decides which function to use.

```
/*
*Routine:  fv_table_name_to_number
*
*Description:  Return the number of a named table
*
*Arguments:  table_name -- (IN) : the name of the table
*            table_number -- (OUT) : the number of the table
*
*Return Value:  Error code
*
*Errors:  fv_table_name_undefined
*
*Side Effects:  Assignment to table_number
*/
fv_error_type fv_table_name_to_number(char *table_name, int *table_number);
```

In the pair of routines `fv_`*table*`_index_to_string` and `fv_`*table*`_string_to_index` below, we use *table* (in italics) to indicate that one pair of routines exists for each field value table defined in the system. When one of the first two routines above is called, it simply uses the table number argument to decide which specific table routine to call. The table routines either work by doing a table look-up or by calling a function. For example, the `english_month_name` table routines use a table with twelve entries, and the `day_of_year` table routines use a pair of functions for computing day and index numbers.

```
/*
*Routine:  fv_table_index_to_string
*
*Description:  Return the string in table corresponding to an index value
*
*Arguments:  index -- (IN) : the index value to convert to a string
*            result -- (OUT) : the requested string
*
*Return Value:  Error code
*
*Errors:  fv_index_too_large
*         fv_index_too_small
*         fv_string_overflow
*
*Side Effects:  Assignment to passed result string
*/
fv_error_type fv_table_index_to_string(int index, string_struct result);


/*
*Routine:  fv_table_string_to_index
*
*Description:  Return the index value in table of a string
*
*Arguments:  string -- (IN) : the string to be located
*            result -- (OUT) : the index value of the string
*            num_char -- (OUT) : the number of recognized characters in the string
*
*Return Value:  Error code
*
*Errors:  fv_string_unrecognized
*
*Side Effects:  Assignment to result and num_char
*/
fv_error_type fv_table_string_to_index(char *string, int *result, int *num_char);
```

8.3 Internal Data Structures

The `fv_table_name_to_number` routine uses the first array defined below for finding the number of a field value table. The numbers of genuine tables range from 0 to `fv_max_field_tables` -1, and the numbers of functions implementing field value tables range from `fv_max_field_tables` to `fv_max_field_tables` + `fv_max_field_functions` -1.

```
#define fv_max_field_tables num /*total number of genuine field value tables, generated */
#define fv_max_field_functions num /*total number of field value functions, generated */

char *table_names[fv_max_field_tables + fv_max_field_functions - 1];
```

The data structure below represents genuine field value tables. The first three fields are used for output, i.e., when an index value is looked up and the corresponding field value is returned. The fourth field (an array of `struct`'s) is used for input, i.e., when a field value must be transformed into an index value. The latter array is sorted on the field values.

```
struct {
          int min_index;
          int max_index;
          char *field_values[];
          struct { char *field_value; int index_value; } input_table[];
} field_value_tables[fv_max_field_tables];
```

The data structure immediately following represents the pairs of field value functions.

```
struct {
          fv_error_type (*fv_table_index_to_string)(int, string_struct);
          fv_error_type (*fv_table_string_to_index)(char *, int *, int *);
} field_value_functions[fv_max_field_functions];
```

9 Examples

By means of examples, we illustrate how components of the architecture interact when performing temporal constant related operations.

9.1 Adding Variable Spans and Events

Consider adding the variable span `%month%` to the event `|January 27, 1992, 12:01 PM|`. The result of adding a month and an event is, in the calendar of this example, defined to be the event of the following month which is the same number of days from the month's end as was the argument event. In our example, January 27 is 4 days from the end of January. Thus, the result is 4 days from the end of February, or `|February 25, 1992, 12:01 PM|` (1992 is a leap year).

In the architecture, it is first determined that the span is variable. The TADT routine `tadt_is_vs_s` is used. Then the routine `cal_`*calendar*`_vs_add_e` is called with the event and the variable span, month. It, in turn, calls `tadt_e_to_seconds` which returns the number of seconds from the argument event to the origin. The calendar routine then uses this number to compute the month, day and year represented by the event (i.e., January 27, 1992). The month is then determined from this information; the desired result after the addition is determined to be February 25, 1992, 4 days from the end of February. The calendar routine then computes how many seconds from the temporal origin this time and date result is. Finally, it calls the routine `tadt_seconds_to_e` which returns the result event. No new structures to hold time-stamps would be created here.

Whenever the UCS operates on spans, it asks the TADT module whether the span is variable. While arithmetic operations dealing with fixed spans are handled solely by the UCS and TADT modules, variable spans require calendar involvement—only the calendar, a variable span belongs to, is capable of computing its value.

9.2 Span Arithmetic

Consider an example where the query processor encounters the expression `%age% + %seniority%`. Here `age` and `seniority` are attribute names of type span. In response, the query processor makes the following function call.

```
ucs_s_add_s(age, seniority)
```

Note that the query processor is not capable of determining whether the spans are fixed or variable. Therefore, the routine `ucs_s_add_s` is called independently of the types of the spans involved—this is the routine that supports addition of span values in the type system of the query language. The routine determines whether its operands are fixed, variable, or a combination. To do so, it calls the routine `tadt_is_vs` as follows.

```
tadt_is_vs_s(age)
tadt_is_vs_s(seniority)
```

If both `age` and `seniority` are fixed spans, the following function call is made.

```
tadt_fs_add_fs(seniority, age)
```

In this case, the TADT performs the addition. If exactly one of the attributes, e.g., `seniority`, is a variable span then a calendar is responsible for the computation. The UCS issues the function call

```
cal_calendar_vs_add_fs(seniority, age)
```

Next, if both spans are variable then the operation is passed to a calendar using the similar routine `cal_`*calendar*`_vs_add_vs`.

Finally, the result computed by one of the routines `tadt_fs_add_fs`, `cal_`*calendar*`_vs_add_fs`, or `cal_`*calendar*`_vs_add_vs` is returned to `ucs_s_add_s` which then returns it to the query processor.

In the example, we only considered addition. The UCS also provides routines for all other arithmetic operations involving span values.

In general, when a computation involves both a variable span and a fixed span then both spans are passed to the calendar of the variable span. The calendar then assumes the responsibility for the computation. It may proceed in several ways. It may convert the variable span to a fixed span and then use the TADT arithmetic routines for performing the computation. It may ask the TADT to determine the size of the fixed span (the difference between its starting and ending events), and decide how to proceed. Other types of operations are also possible.

As illustrated by the example, when an operation involves two variable spans from the same calendar then the UCS simply calls the required routine `cal_vs_`*operation*`_vs` (where *operation* is the mathematical operation, e.g., `add`). If the variable spans are from different calendars then the UCS calls each of the two calendar's routine `cal_vs_to_fs`. The two fixed spans returned from the function calls are used by the TADT to compute the result. Note that adding variable spans from different calendars may produce unexpected results.

Comparisons involving variable spans are similar to arithmetic computations involving variable spans, like that of the example.

9.3 Aggregate Computation

Consider the computation of the average of a set of spans. For the purpose of this example, we assume that the set contains both fixed and variable spans, and that at least one fixed span is evaluated before a variable span is encountered.

Computing aggregates on variable spans requires action by the underlying calendar upon which the spans are defined. Initially the routine `tadt_init_avg_fs` is called to initialize the state used. Then the routine `tadt_accum_avg_fs` is called repeatedly with fixed span arguments until the first variable span is encountered (before each call, the routine `tadt_is_vs_s` is used to decide the type of the span). Now the UCS calls the calendar's routine `cal_`*calendar*`_avg_fstate_to_vstate` which converts the partial state of fixed spans to a state usable for variable span average computation. Then the routine `cal_`*calendar*`_accum_avg_vs` is called with each of the rest of the spans, independently of their type. Finally, `cal_`*calendar*`_final_avg_vs` is called after all of the spans have been processed.

9.4 Property Table Example

The use of properties adds flexibility to the calendars, and it helps tailoring the calendars to the needs of different users. The properties may be used to specify the input and output formats of temporal constants, the current locale, an override input epoch, and the naming of the concepts beginning and forever.

The calendar DBMS is created with an initial property table, containing values for each of the ten properties. A new property table is created as follows. First the new values of the properties to be changed are specified using the `ucs_property_push` routine. Second, the new table is created by calling the routine `ucs_property_activate`. The new table is identical to the previous table, except from that it contains the new values specified with the `ucs_property_push` routine. If no new value is specified for a property, the old value is in effect. If more than one value is specified for the same property, the most recently specified value is in effect.

Using the routine `ucs_property_deactivate`, the effect of the most recent `ucs_property_activate` may be undone. In this way it is possible to revert to any previous table. To exemplify, suppose that our application is target for use in the European Economic Community, where multi-lingual reports on information in the database need to be generated. Most of the information must be displayed in English, but selected portions must be presented in other languages, e.g., French, Danish.

Initially, the *Event Output Format* property has the following value.

`<month,english_month_names>␣<day,arabic_numeral>,␣<year,arabic_numeral>`

This prints out the *Ides of March* in 1992 as `March 15, 1992`. Assume that we need to print this and other dates according to the common European format (day/month/year). To do so, we change the initial event output format to the following format.

`<day,arabic_numeral>/<month,arabic_numeral>/<year,arabic_numeral_two_digits>`

This format is specified using the function `ucs_property_push`. The new table is created by calling `ucs_property_activate`. The actual sequence of calls follows.

```
ucs_property_push("event output format", "day/month/year property string");
                        ucs_property_activate();
```

Now assume that the new dates must be printed in France. Then we may want to specify a new value for the *Locale* property. The value, indicating some location in France, may be simply the character string for "`France`", or it may be a longitudinal coordinate specifying a point in France. The *Locale* property value is interpreted by the calendar. A new table (still with the day/month/year output format) is created as follows.

```
ucs_property_push("locale", "France");
        ucs_property_activate();
```

The *Ides of March* would now be printed out as `15/3/92`.

After printing out a few dates in this format, suppose one date needs to be printed out using French month names. We specify the following format.

`<day,arabic_numeral>␣<month,french_month_names>,␣<year,arabic_numeral>`

To create the appropriate table, we issue these commands.

```
ucs_property_push("event output format", "French month names format");
                        ucs_property_activate();
```

Our earlier date would be printed out as `15 Mars, 1992`. The new date is still in France, so the "`France`" *Locale* property value is still desired. Since we did not explicitly change the old *Locale* property value, the new table contains the desired value. After this date is printed, we use the routine `ucs_property_deactivate` to revert to toe previous property table, and the day/month/year event output format becomes current. Finally, after the use of this property is no longer needed, the original property table is recovered using `ucs_property_deactivate`.

Situations may arise where the runtime system uses the `ucs_property_push` routine repeatedly and later discovers that an erroneous property value has been pushed. To recover easily from such a mistake, we provide the routine `ucs_property_clear`. Instead of calling the routine `ucs_property_activate` and then `ucs_property_deactivate`, this new routine may be called. Its effect is to simply undo all invocations of `ucs_property_push` after the most recent invocation of `ucs_property_activate`.

9.5 Time-stamp and Temporal Constant Translation

When a temporal constant is read, one or more calendars may need to be consulted to translate the constant. Calendars are consulted in order of the calendar priority list until a calendar successfully translates the constant. The property *override input epoch* may be used to override the default input priority list and this epoch (and associated calendar) is considered first for translation of the constant.

Input format properties are provided for each of the three temporal types. We have designed the query language to include strong typing of temporal expressions, including temporal constants. The query processor is therefore able to determine at compile time which UCS translation function to call. The UCS determines the format string for the particular temporal type and translates the constant according to the contents of this string. The calendar then fills a table with information from the parsed input string.

The calendar knows how to convert this into the number of seconds from the origin (using internal rules defined in the implementation of the calendar). The calendar calls the TADT to create an event, interval, or span and return this data structure to the calendar. The calendar does not understand the internals of this structure; whenever it is necessary to understand the internal contents the TADT module is called to convert the structure back to the number of seconds from the origin.

Time-stamp translation and constant translation are discussed in detail in the following two examples.

9.5.1 Time-stamp To String Translation

Figure 3 shows a flow diagram for the processing that occurs when a time-stamp is converted into an output string. (This processing would occur when a time-stamp is retrieved in a `fetch` statement returning temporal attributes.) The query processor invokes the UCS to convert the retrieved time-stamp into an output string for assignment to a procedure parameter. Boxes in the figure denote actions which, in turn, represent UCS or calendar calls. Ovals represent data items used in or generated by the processing. Most actions present in the figure are implemented in the UCS; calendar routines are represented by broken outline boxes, and we note that there are only two such boxes.

Figure 3 is illustrative of how table-driven algorithms are used in the UCS. Consider the time-stamp `000078772A800000` stored in the `when_employed` attribute of Figure 1. This is the actual time stamp for midnight, January 2, 1972 MST, as a hexadecimal number in the high resolution representation. For this time-stamp, the `american` calendric system is consulted, and the time-stamp is determined to be associated with the `gregorian` calendar. Translation begins by performing local processing to determine the correct timezone. The calendar checks the value of the *Locale* property which names the location of interest. In this case, the locale is "`Tucson, Arizona`". The calendar uses internal tables or computation to determine the timezone in which Tucson is located, or its offset from Greenwich Mean Time (GMT). The locale indicates that Tucson, and most of the state of Arizona, is always on Mountain Standard Time (MST), making determining the timezone particularly simple. The time displacement for MST, which is 7 hours behind GMT, is retrieved and subtracted from the original time-stamp. All of the parsing of the locale is internal to the calendar.

Generation of the final output string begins by invoking the `gregorian` calendar to generate the following *array of field values*. The UCS uses the valid flags to indicate which array entries are desired.

Index	*Value*	*Valid*
0	2	Y
1	0	Y
2	1972	Y
...	...	...

The array of field values is simply an unparsed version of the time-stamp. The content of the array of field values is described by the following *field index table*, which is provided by the calendar.

Index in Field Value Table	*Field Name*
0	`day`
1	`month`
2	`year`
...	...

Table 5: Field Index Table for Gregorian Calendars

The *field index table* associates indices in the array of field values with the components of a temporal constant. The field index table indicates that the day component, 2, is found in the zeroth element of the array of field values, the month component, 0, is found in the first element, and the the year component, 1975, is found in the second element.

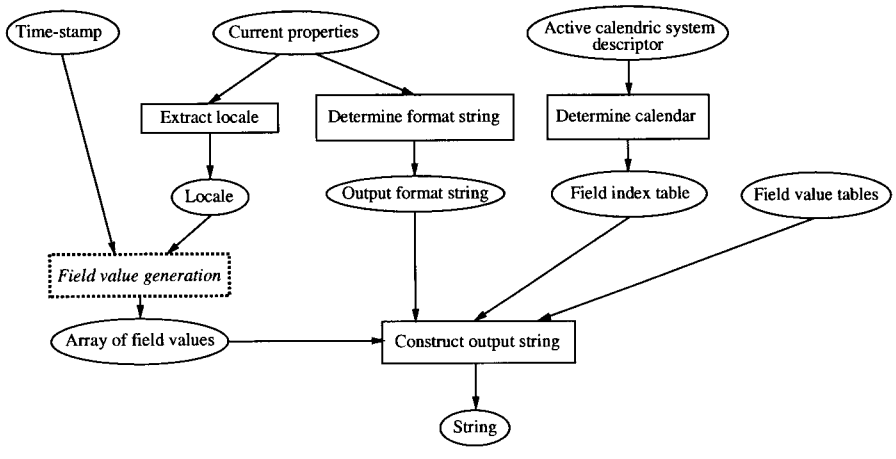

Figure 3: Time Value Retrieval

The UCS determines the format of the final output string by retrieving the *output format string* from the current property set. We schematically represent the output format string as follows.

`<month,english_month_names>␣<day,arabic_numeral>,␣<year,arabic_numeral>`

The format string lists the fields in the order that they are to appear in the output. The component associated with each field is either a *field value name table* or the name of a routine that computes the field's string. For example, the `english_month_names` field value name table is shown in Table 6. The UCS retrieves the string `January` since the field value table entry for `month` is 0, and `January` is contained in the zeroth entry of the field value name table. The UCS iterates over

the fields of the output format string adding one field to the output string on each iteration. The resulting string, "`January␣2,␣1975`", is returned as the value of the `when_employed` attribute.

Index	*Field Value*
0	`January`
1	`February`
...	...
10	`November`
11	`December`

Table 6: `english_month_names` Field Value Table

This example illustrates how much of the the processing has been moved out of the calendar and into the UCS and TADT. In particular, the calendar need only provide one routine, that converts an adjusted time-stamp into an array of field values. (The calendar may also define additional field formatting routines as previously described.) The UCS does the rest of the work of creating the associated string. Minimizing each calendar's responsibility is important since the UCS and TADT will be implemented once, by the DBMS implementor, whereas the calendar's implementation will be the responsibility of the DBI.

9.5.2 String To Time-stamp Translation

Figure 4 shows the flow diagram for temporal constant interpretation. This flow occurs when the string denoting a temporal constant is translated during run-time.

Consider reading in the string "`January␣2,␣1972`". The query processor begins the parsing of this time stamp by calling the UCS with the string. The UCS looks at the priority table and the property table to determine which calendar has first priority when translating this string. These calendars are consulted in order until one succeeds in translating the constant. For this constant, the Gregorian calendar will succeed.

We'll use the same format as before, only this time for input. When the string is parsed according to this format, it results in the values shown in Table 7. Here the valid column indicates whether the associated field appeared in the input format.

Index	*Value*	*Valid*
0	`2`	Y
1	`0`	Y
2	`1972`	Y
...	...	N

Table 7: Result of Parsing "`January 2, 1972`"

The calendar must still convert the parsed time into a number, which represents seconds or fractions of seconds from the origin. This number represents the input string, but it does not represent a universal time. It represents a time which is relative to its *locale*. Locale is a property from the UCS property table which is calendar dependent. This value could describe a simple time zone, such as `MST`, it could give the name of the city such as `La Paz`, or it could even give the longitude of the locale. Whatever information is contained in the *locale* property, the calendar must be able to convert this into a time offset from Greenwich Mean Time.

This offset is then applied to the number computed above, and this final number is passed to the

TADT module which creates the event (Midnight January 2, 1972 is the hex value 000078772A800000) which represents the input string.

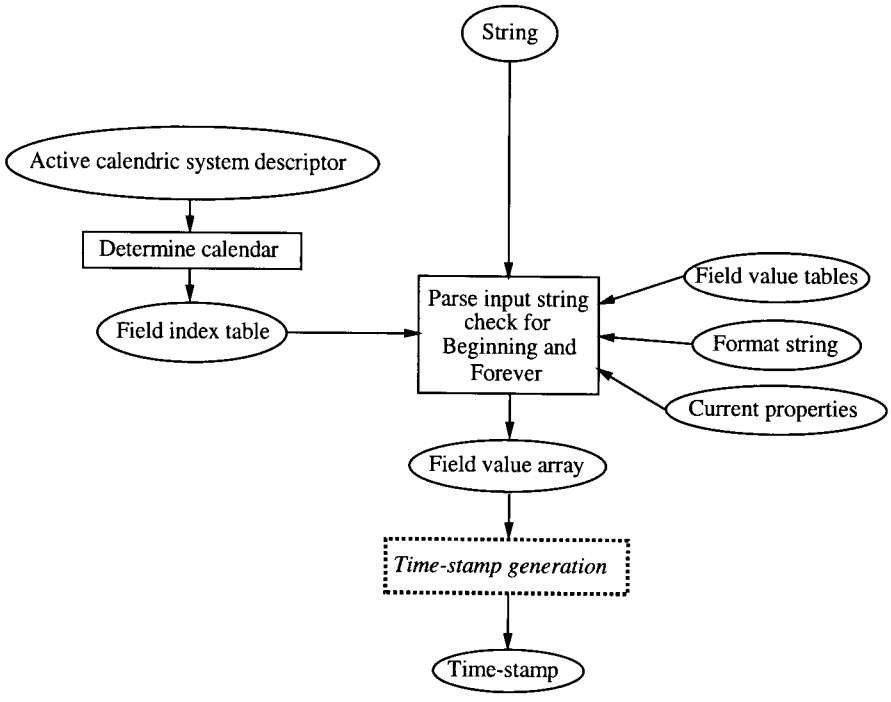

Figure 4: Time Value Interpretation

10 Calendar DBMS Generation Toolkit

The architecture described in this document shares the characteristics of most extensible DBMSs, in that certain aspects are bound at DBMS-generation time, other aspects are bound at database schema-definition time, and still other aspects are bound during query evaluation. Specifically, in our design calendars and calendric systems are declared when the DBMS is generated; calendric systems are bound at schema definition time (or more precisely, when an SQL module is compiled), and properties, such as output format, are bound at query evaluation time.

In this section, we focus on the aspects that are bound when the calendar DBMS is generated. The calendar DBMS generation toolkit processes a number of user-made specifications and generates data structures used by the UCS and the FV modules.

Figure 5 gives an overview of the process of generating a DBMS, and it illustrates the role of the toolkit, which consists of four individual generators. The top row of ovals represent high-level, user provided specifications. The row of boxes are individual generators within the toolkit. Each of these generate C language output. This output, along with C code defining the TADT module, a library of UCS routines, and the remaining parts of a DBMS, is compiled to generate a DBMS supporting multiple calendars.

We proceed by describing the input to the toolkit and its individual generators. Then we discuss the products generated by the toolkit. For each product, we explain briefly what input is used to generate it.

The user-made specifications read by the toolkit are contained in several types of specifications.

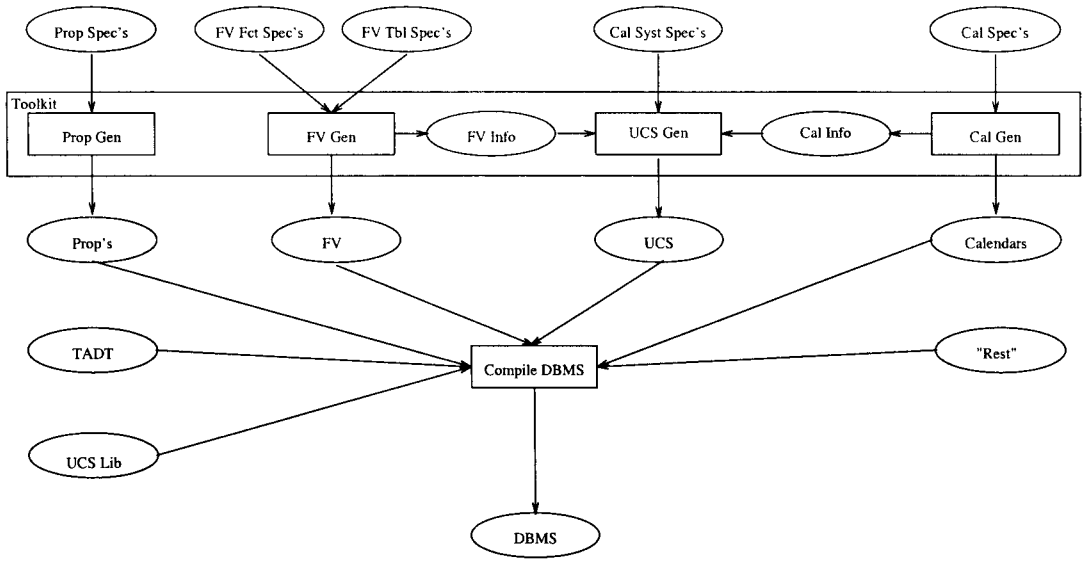

Figure 5: Using the Toolkit for Generating a DBMS

Prop Spec's—property specifications As described in Section 6.2.5, there are ten types of properties. This default property specification contains a single value for each of these, and it is used by the default property generator (Prop Gen).

FV Fct Spec's—field value function specifications The size of a domain of field values, e.g., `arabic_numeral`, may make enumeration impractical. When that is the case, field value functions are utilized. A table is represented by two functions, one maps index values to field values, and the other maps field values to index values. While a higher level specification language is desirable, field value functions must currently be specified using the C programming language. See also Section 8.2.

FV Tbl Spec's—field value table specifications Table 6 is an example of this kind of table. Field value tables enumerate domains of field values such as `english_month_names` and `danish_week_days`. Section 8.3 discussed these tables.

FV Info—field value information The UCS generator needs a listing of all available field value tables and functions. This listing is compiled from the field value specifications, by the FV generator.

Cal Syst Spec's—calendric system specifications This specification describes the various calendric systems to be used by the DBMS. For each calendric system, its name and a list of calendars, each with an associated epoch, is given. The calendars are ranked for the purpose of interpretation of input strings. Starting with the highest ranked calendar, an input string is passed to successively lower ranked calendars until it is recognized. The output generated by the UCS generator is described in Section 6.3.

Cal Info—calendar information The calendar generator produces a list of signatures of all calendar routines, as discussed in Section 6.2.8. This list is used by the UCS Generator (UCS Gen).

Cal Spec's—calendar specifications These specifications are used by the calendar generator (Cal Gen) to define the calendric systems included in the DBMS. Section 7 discusses calendars

and calendar-dependent operations in particular. The preliminary version of the toolkit does not construct the C routines of calendars from higher-level specifications—the routines must be supplied directly. (See also Section 6.3.)

A central goal is to make the specification of calendars as easy as possible. Therefore, we want to improve the toolkit so that it can generate calendar routines in the C programming language from declarative descriptions. Already, we have simplified the specification of calendars by shifting as much responsibility as possible from individual calendars to system components that are not user-specified.

As seen in Figure 5, the toolkit generators produce four components. We discuss each of these components in turn.

Prop's—properties The property stack is the topic of Section 6.2.5. The Property generator creates the initial property stack, `initial_property_list` (Section 6.3).

FV—field value module In addition to field value tables and functions (`field_value_tables` and `field_value_functions`), this module contains a table, `table_names`, that relates table names (including function names) to table numbers. These data structures are described in Section 8.3.

UCS—uniform calendric support A data structure, `calendric_system_list`, enumerates the calendric systems defined for the DBMS. For each calendric system, this data structure gives its name and lists its constituent calendars along with their epochs. The position in this list defines the rank of a calendar when interpreting input strings. This and the following structure is defined in Section 6.3. The global function table, `calendar_functions`, contains the signatures of each function supplied by some calendar. Because the list of functions supported by a calendric system is the union of the set of functions defined by each calendar, function names must be unique within a complete calendric system. A list of all field value tables and functions is included.

Calendars The calendar component of the architecture mainly consists of calendar dependent routines. The toolkit verifies that all required routines, as described in Section 7, are present. In addition, a calendar contains a field index table which is a component of `calendric_system_list` (Section 6.3), which is used when a stored time stamp is translated to or from a text string. Its use is exemplified in Section 9.5.

Note that the TADT is not at all affected by user specifications.

11 Future Work

There are still several areas that need to be further considered before this design is complete. Historical indeterminacy is only partially accommodated. While the various routines may take indeterminate events, intervals, and spans as arguments, and return such time-stamps, full support for historical indeterminacy entails either additional arguments specifying ordering plausibility and range credibility, or additional routines where these values are not assumed to both be 100.

Second, the input routines discussed in Section 9.5.1 assume that a single format is available for input. This should be generalized to allow variable-format input.

Finally, the Cal Gen tool in the generation toolkit has not been designed. In the interim, we plan to manually produce the C source code for the calendars, but obviously a specialized high-level calendric specification language would be preferable.

Acknowledgements

Suchen Hsu contributed to previous drafts of this proposal. Support for this research was provided in part by the National Science Foundation through grant IRI-8902707 and by the IBM Corporation through contract #1124. Christian S. Jensen was in addition supported by Aalborg University, Denmark, and by Danish Natural Science Research Counsil through grant 11-9675-1 SE.

Bibliography

[Dyreson & Snodgrass 1992] Dyreson, C. E. and R. T. Snodgrass. "Time-stamp Semantics and Representation." TempIS Technical Report 33. Computer Science Department, University of Arizona. Revised Apr. 1992, 41 pages.

[Soo & Snodgrass 1992A] Soo, M. and R. Snodgrass. "Mixed Calendar Query Language Support for Temporal Constants." TempIS Technical Report 29. Computer Science Department, University of Arizona. Revised May, 1992, 59 pages.

[Soo & Snodgrass 1992B] Soo, M. and R. Snodgrass. "Multiple Calendar Support for Conventional Database Management Systems." Technical Report 92-7. Computer Science Department, University of Arizona. Feb. 1992.

Exported Routine Prototypes

```
max_state_vs     cal_allocate_max_state_vs();                                  43
min_state_vs     cal_allocate_min_state_vs();                                  43
sum_state_vs     cal_allocate_sum_state_vs();                                  43
tadt_error_type tadt_abs_fs(span_type, span_type)                              10
error_type       tadt_accum_avg_e(avg_state_e, event_type)                     19
error_type       tadt_accum_avg_fs(avg_state_fs, span_type)                    19
error_type       tadt_accum_count_e(int *, event_type)                         19
error_type       tadt_accum_count_fs(int *, span_type)                         19
error_type       tadt_accum_count_i(int *, interval_type)                      19
error_type       tadt_accum_max_e(event_type, event_type)                      19
error_type       tadt_accum_max_fs(span_type, span_type)                       19
error_type       tadt_accum_min_e(event_type, event_type)                      19
error_type       tadt_accum_min_fs(span_type, span_type)                       19
error_type       tadt_accum_sum_fs(span_type, span_type)                       19
avg_state_e      tadt_allocate_avg_state_e()                                    8
avg_state_fs     tadt_allocate_avg_state_fs()                                   8
event_type       tadt_allocate_e(resolution_type)                               8
interval_type    tadt_allocate_i(resolution_type, resolution_type)              8
string_struct    tadt_allocate_string_struct()                                  8
span_type        tadt_allocate_s(resolution_type)                               8
timestamp_type   tadt_allocate_ts()                                             8
vs_type          tadt_allocate_vs_struct()                                      8
tadt_error_type tadt_begin(interval_type, event_type)                           8
bool             tadt_can_fit_ts(timestamp_type, int)                          21
tadt_error_type tadt_coerce_e(event_type, int, event_type)                     21
tadt_error_type tadt_coerce_fs(span_type, int, span_type)                      22
tadt_error_type tadt_coerce_i(interval_type, int, interval_type)               22
error_type       tadt_create_e(seconds_type, resolution_type, timestamp_type)  19
error_type       tadt_create_fs(seconds_type, resolution_type, timestamp_type) 20
error_type       tadt_create_i(event_type, event_type, timestamp_type)         20
error_type       tadt_create_vs(vs_type, span_type)                            20
error_type       tadt_e_add_fs(event_type, span_type, event_type)              11
bool             tadt_e_equals_e(event_type, event_type)                       15
error_type       tadt_e_minus_e(event_type, event_type, span_type)             12
error_type       tadt_e_minus_fs(event_type, span_type, event_type)            12
tadt_error_type tadt_end(interval_type, event_type)                             8
bool             tadt_e_overlaps_i(event_type, interval_type)                  16
bool             tadt_e_precedes_e(event_type, event_type)                     15
bool             tadt_e_precedes_i(event_type, interval_type)                  15
error_type       tadt_e_to_seconds(event_type, seconds_type)                   22
error_type       tadt_extract_vs(span_type, vs_type)                           20
error_type       tadt_final_avg_e(avg_state_e, event_type)                     19
error_type       tadt_final_avg_fs(avg_state_fs, span_type)                    19
error_type       tadt_final_count_e(int, int *)                                19
error_type       tadt_final_count_fs(int, int *)                               19
error_type       tadt_final_count_i(int, int *)                                19
error_type       tadt_final_max_e(event_type, event_type)                      19
error_type       tadt_final_max_fs(span_type, span_type)                       19
error_type       tadt_final_min_e(event_type, event_type)                      19
error_type       tadt_final_min_fs(span_type, span_type)                       19
error_type       tadt_final_sum_fs(span_type, span_type)                       19
tadt_error_type tadt_first(event_type, event_type, event_type)                 10
bool             tadt_free(void *)                                              8
error_type       tadt_fs_add_fs(span_type, span_type, span_type)               11
```

```
error_type      tadt_fs_div_fs(span_type, span_type, double *)                    13
error_type      tadt_fs_div_n(span_type, double, span_type)                       13
bool            tadt_fs_eq_fs(span_type, span_type)                               14
bool            tadt_fs_gt_fs(span_type, span_type)                               15
bool            tadt_fs_lt_fs(span_type, span_type)                               14
error_type      tadt_fs_minus_fs(span_type, span_type, span_type)                 11
error_type      tadt_fs_times_n(span_type, double, span_type)                     12
error_type      tadt_i_add_fs(interval_type, span_type, interval_type)            13
bool            tadt_i_contains_i(interval_type, interval_type)                   17
bool            tadt_i_equals_i(interval_type, interval_type)                     16
bool            tadt_i_meets_i(interval_type, interval_type)                      17
error_type      tadt_i_minus_fs(interval_type, span_type, interval_type)          13
avg_state_e     tadt_init_avg_e()                                                 19
avg_state_fs    tadt_init_avg_fs()                                                19
int             *tadt_init_count_e()                                              19
int             *tadt_init_count_fs()                                             19
int             *tadt_init_count_i()                                              19
event_type      tadt_init_max_e()                                                 19
span_type       tadt_init_max_fs()                                                19
event_type      tadt_init_min_e()                                                 19
span_type       tadt_init_min_fs()                                                19
span_type       tadt_init_sum_fs()                                                19
tadt_error_type tadt_intersect(interval_type, interval_type, interval_type)        9
tadt_error_type tadt_interval(event_type, event_type, interval_type)               9
bool            tadt_i_overlaps_i(interval_type, interval_type)                   17
bool            tadt_i_precedes_e(interval_type, event_type)                      16
bool            tadt_i_precedes_i(interval_type, interval_type)                   16
bool            tadt_is_vs(span_type)                                             21
tadt_error_type tadt_last(event_type, event_type, event_type)                     10
void            *tadt_malloc(int)                                                  7
error_type      tadt_neg_fs(span_type, span_type)                                 11
tadt_error_type tadt_present(event_type)                                          10
error_type      tadt_seconds_to_e(event_type, seconds_type)                       23
tadt_error_type tadt_span(interval_type, span_type)                                9
error_type      ucs_accum_avg_s(ucs_avg_state_s avg_state, span_type span);        34
error_type      ucs_accum_count_s(int *count_state, span_type span);               34
error_type      ucs_accum_max_s(ucs_max_state_s max_state, span_type span);        34
error_type      ucs_accum_min_s(ucs_min_state_s min_state, span_type span);        34
error_type      ucs_accum_sum_s(ucs_sum_state_s sum_state, span_type span);        34
ucs_avg_state_s ucs_allocate_avg_state_s();                                        30
ucs_max_state_s ucs_allocate_max_state_s();                                        30
ucs_min_state_s ucs_allocate_min_state_s();                                        30
ucs_sum_state_s ucs_allocate_sum_state_s();                                        30
ucs_error_type  ucs_bind_function(char *, int, formal_arg_type, formal_arg_type *,
                          generic_function_handle)                                39
error_type      ucs_declare_calendric_system(char *)                              40
error_type      ucs_e_add_s(event_type, span_type, event_type)                    31
error_type      ucs_e_minus_s(event_type, span_type, event_type)                  31
ucs_error_type  ucs_e_string_to_e(char *, event_type)                             37
error_type      ucs_e_to_string(event_type, string_struct)                        37
error_type      ucs_final_avg_s(ucs_avg_state_s final_state, span_type final_avg_span);  34
error_type      ucs_final_count_s(int final_state, int *final_count);             34
error_type      ucs_final_max_s(ucs_max_state_s final_state, span_type final_max_span);  34
error_type      ucs_final_min_s(ucs_min_state_s final_state, span_type final_min_span);  34
```

Exported Types