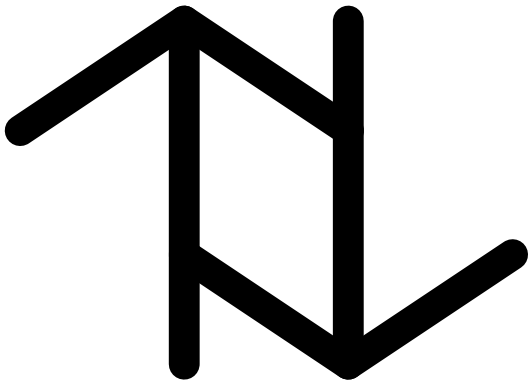

Integrating Multiple Calendars using τZaman

Bedirhan Urgun, Curtis E. Dyreson, Nick Kline, Jessica K. Miller, Richard T. Snodgrass, Michael D. Soo, and Christian S. Jensen

August 19, 2004

TR-80

A TIMECENTER Technical Report

Title	Integrating Multiple Calendars using τ**Zaman**
Author(s)	Bedirhan Urgun, Curtis E. Dyreson, Nick Kline, Jessica K. Miller, Richard T. Snodgrass, Michael D. Soo, and Christian S. Jensen
Publication History	August 2004. A TIMECENTER Technical Report

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Michael H. Böhlen, Heidi Gregersen, SimonasŠaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Dengfeng Gao, Bongki Moon, Sudha Ram

Individual participants
Curtis E. Dyreson, Washington State University, USA; Fabio Grandi, University of Bologna, Italy; Vijay Khatri, Indiana University, USA; Nick Kline, Microsoft, USA; Gerhard Knolmayer, University of Bern, Switzerland; Thomas Myrach, University of Bern, Switzerland; Kwang W. Nam, Chungbuk National University, Korea; Mario A. Nascimento, University of Alberta, Canada; John F. Roddick, Flinders University, Australia; Keun H. Ryu, Chungbuk National University, Korea; Dennis Shasha, New York University, USA; Michael D. Soo, amazon.com, USA; Andreas Steiner, TimeConsult, Switzerland; Paolo Terenziani, University of Torino; Vassilis Tsotras, University of California, Riverside, USA; Jef Wijsen, University of Mons-Hainaut, Belgium; and Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.auc.dk/TimeCenter>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

Programmers world-wide are interested in developing applications that can be used internationally. Part of the internationalization effort is the ability to engineer applications to use dates and times that conform to local calendars yet can inter-operate with dates and times in other calendars, for instance between the Gregorian and Islamic calendars. τZAMAN is a system that provides a natural language and calendar-independent framework for integrating multiple calendars. τZAMAN performs "runtime-binding" of calendars and language support. A running τZAMAN system dynamically loads calendars and language support tables from XML-formatted files. Loading a calendar integrates it with other, already loaded calendars, enabling users of τZAMAN to add, compare, and convert times between multiple calendars. τZAMAN also provides a flexible, calendar-independent framework for parsing temporal literals. Literals can be input and output in XML or plain text, using user-defined formats, and in different languages and character sets. Finally, τZAMAN is a client/server system, enabling shared access to calendar servers spread throughout the web. This paper describes the architecture of τZAMAN and experimentally quantifies the cost of using a calendar server to translate and manipulate dates.

Keywords: Time, multiple calendars, calendric systems, temporal data types, datetime representation.

1 Introduction

There is a need for a system that can support multiple calendars. Temporal data is present in some form in most applications. Einstein's theory of relativity posits that an observer measures time relative to a frame of reference. For most observers, especially those traveling at a (small!) fraction of the speed of light, the frame of reference is influenced most by the observer's cultural and linguistic background. Diverse backgrounds have produced many different ways to measure time. According to Fraser, about forty major calendars are in daily use [Fra87]. Even though time is measured, represented, and used in many different ways, most applications impose a single interpretation for time and temporal operations. For instance, the SQL-92 standard database query language requires dates to be represented solely in the Gregorian calendar [MS93].

This paper presents τZAMAN, a system that provides temporal functionality for applications that need to calculate, format, parse, and/or compare times within either a single calendar or across multiple calendars. The project name is composed of the Turkish word for time, *Zaman*, (pronounced "Zah-mon"), and the Greek letter, τ (pronounced *tau*), which denotes that it is part of the Temporal Access for Users (tau) project started at the University of Arizona.[1]

The intended use of τZAMAN is as a calendar server, for multiple calendars. τZAMAN takes a "runtime-binding" approach to integrating multiple calendars. In runtime binding, calendars and supporting tables are developed in isolation at different locations, and are subsequently loaded as needed into a running τZAMAN system. For instance, a developer in France could specify a Gregorian calendar, another in Australia could write tables for month names in English, a third developer in Saudi Arabia could build an Islamic calendar, and a fourth in Japan could write Islamic month names in Japanese. Each developer works independently. When finished, a developer places a description of his or her work on the web formatted in the Extensible Markup Language (XML) [W3C00]. Then a user in Canada could specify a *calendric system* utilizing all of these resources though a simple specification (again in XML). τZAMAN integrates the calendars only when the calendric system is loaded. Users of the system can input and output times in different languages and calendars, perform inter-calendar conversions, and compare and modify times as desired. τZAMAN also provides a range of arithmetic and comparison operations on times, for example there is an operation to add an interval (e.g., "1 week") to an instant (e.g., "January 1, 2004").

[1] `http://www.cs.arizona.edu/tau`

τZAMAN is a calendar-independent framework that incorporates several novel features for enabling the rapid integration of multiple calendars.

- τZAMAN is a client/server system. Calendars can be complicated and costly to develop, which is one reason why applications usually have limited support for time. When a calendar is developed, it is useful to share the calendar among many applications and users. A client/server system enables the creation of "calendar servers" that can provide calendar-related services to multiple clients. We anticipate that there will be τZAMAN servers, or more precisely τZAMAN web services, running on well-known sites, especially for the major calendars.

- A key part of the design of τZAMAN is the ability to add calendars and input-output formats on the fly, at run-time. New calendars and other user-defined information, such as natural languages or input-output formats for temporal literals, can be integrated into a multi-calendar system without recompiling τZAMAN or even stopping and restarting a τZAMAN server.

- τZAMAN makes extensive use of XML. XML is becoming increasingly popular in web applications for exchanging data and describing services. In τZAMAN, all calendar-related specifications are XML documents. Using XML also helps to improve the parsing of the files for specifying τZAMAN components, making it easier to develop calendars. For instance, a specification file in XML can be validated with an XML schema language, like XML Schema [Fal01]. τZAMAN also supports the construction and use of XML-sensitive formats to input and output temporal literals since we anticipate a future growth in the use of XML to represent times and dates.

This paper is organized as follows. The next section presents several example scenarios showing how τZAMAN can be used. Section 3 introduces the major time-related concepts that are implemented in τZAMAN. The architecture is described in Section 4, which consists of an overview of the major packages and a detailed discussion of the roles of individual classes. We show how developers and users create and use calendars in τZAMAN.

We performed several experiments to measure the efficiency of τZAMAN. The results are reported in Section 5. Section 6 presents a prototype end-user and calendar developer tool, with a Graphical User Interface (GUI), that uses τZAMAN to translate and manipulate dates. The last two sections discuss related research and list the contributions and future directions of this research.

2 Usage Scenarios

This section presents several examples to motivate the utility and functionality of τZAMAN. Each example is a separate scenario. The scenarios become increasingly more sophisticated.

In the first scenario a user, let's call her Leslie, has a long list of banking records timestamped with Gregorian calendar dates. The dates are formatted using a style common in the United States of America (`mm/dd/yyyy`). Leslie is sending the records to Paris, so she would like to convert the dates to a format used in Europe (`dd/mm/yyyy`). Figure 1 shows a concrete example of such a conversion. This conversion is very simple. One could imagine writing a Perl script, or a program in another string processing language, to perform the conversion. τZAMAN can also convert times between formats. To do a format conversion, Leslie would first connect to a Gregorian calendar τZAMAN server, push an *Instant input property* with the USA format, and push an *Instant output property* with the European format. Next, for each date, Leslie would construct an instant (e.g., by calling the Instant class constructor) and subsequently have that instant output itself. The instant would be constructed using the Gregorian calendar and the USA format, but output in the European format.

```
04/08/2003 → 08/04/2003
```

Figure 1: Converting a date from a USA to a European format

```
<date>                              <date>
  <month value = "04" />              <day>08</day>
  <day value = "08" />        →       <month>04</month>
  <year value = "2003" />             <year>2003</year>
</date>                             </date>
```

Figure 2: An XML-based conversion from USA to European date format

The second scenario is similar to the first, but instead of an unstructured text document, Leslie has an XML document. The dates in the document are encoded within `<date>` elements. She would like to do the same kind of conversion, from USA to European format, as illustrated in Figure 2. τZAMAN can also perform XML-sensitive conversions. The conversion uses the same processes as the previous scenario, only the Instant input property and Instant output property would have to change to use the XML-based formats (I/O formats can be specified by users). We anticipate that XML-based conversions will become more common than unstructured text conversions in future.

The third scenario concerns changing the language in which a calendar date is represented. Leslie has a friend in India. She'd like to translate Gregorian calendar dates that include an English month name into a date with the month name given in Hindi, without changing the format as illustrated in Figure 3. τZAMAN supports using different languages and different character sets for *fields* in formats, such as the name of the month. New tables for language support, encoded as XML documents, can be dynamically loaded as needed.

The fourth scenario concerns converting times between calendars. Leslie contacts a business in Cairo to integrate her banking information with Egyptian purchase data. The business asks Leslie to translate each Gregorian calendar date to the corresponding date in the Islamic calendar. Figure 4 illustrates the desired conversion from the Gregorian to the Islamic calendar. The figure renders the Islamic date in English for expository purposes; the language could be translated to Arabic during the conversion in a manner similar to the third scenario.

The fifth scenario features a calendar server to convert a time from a Gregorian to an Islamic calendar. A single τZAMAN system can load several calendars at once and apply inter-calendar conversions. τZAMAN could also be deployed in a distributed system as illustrated in Figure 5. The figure shows a "local" user (in this scenario, the local user is the business in Cairo) running τZAMAN that has a reliable implementation of the Islamic calendar. Leslie runs a "remote" τZAMAN server for the Gregorian calendar. The "client" API for τZAMAN is the same for local and remote servers, so clients do not have to be specialized to manage local and remote services differently. From a client's perspective the only difference between local and

```
<date>                                  <date>
  <month value = "January" />             <month value = "Magha" />
  <day value = "08" />            →       <day value = "08" />
  <year value = "2003" />                 <year value = "2003" />
</date>                                 </date>
```

Figure 3: A time value is translated from English to Hindi

```
<date>                                    <date>
  <month value = "January"/>                <month value = "Safar"/>
  <day value = "08"/>            →          <day value = "06"/>
  <year value = "2003"/>                    <year value = "1424"/>
</date>                                   </date>
```

Figure 4: A Gregorian calendar to Islamic calendar conversion

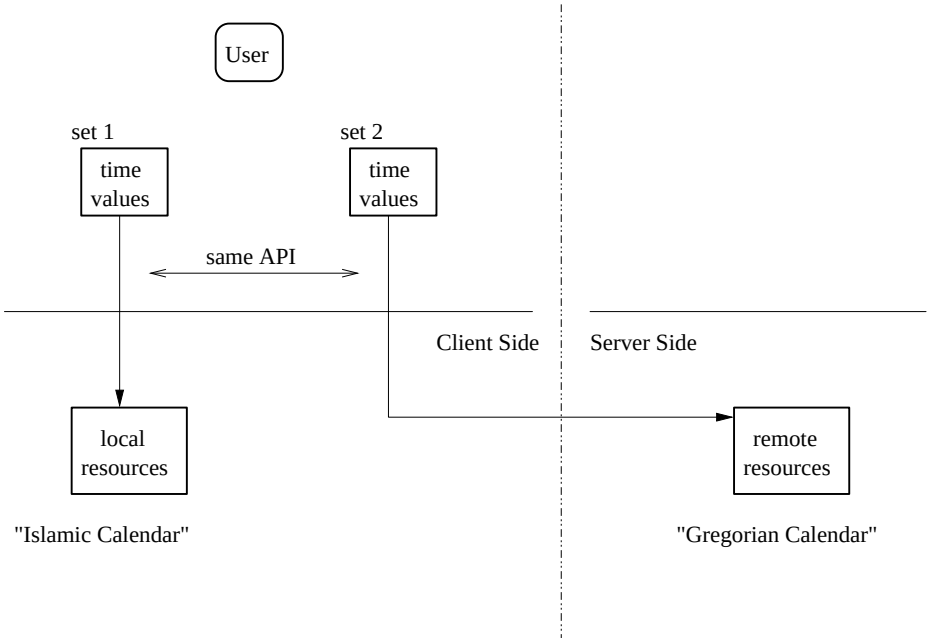

Figure 5: Converting between local and remote τZAMAN servers

remote servers, other than performance, is that the servers have different names. The figure shows a client in contact with a single remote server, but in general, a τZAMAN client can simultaneously communicate with multiple τZAMAN servers.

The sixth scenario examines a time granularity conversion. Suppose Leslie wants to know how much she spends each month. In order to calculate the amount per month, she needs to convert the date of each banking record from a granularity of Gregorian calendar `days` to a granularity of Gregorian `months`, so that she knows which records are in the same month. Figure 6(a) illustrates this simple granularity conversion. A less straightforward conversion would be from `days` to a granularity of Gregorian `weeks` (assuming Leslie would like to do a weekly analysis of her spending). An even more complicated conversion would be converting a time at a granularity of `months` to one at a granularity of `days` (or `weeks`). For example, suppose Leslie knows she bought an item in March 2003, but does not know the exact day when she bought the item. Generally, conversions from coarse to fine granularities result in *indeterminate* times [DS98]. An indeterminate time is a time that is not precisely specified, such as "sometime in March" or "last week". Figure 6(b) shows an example conversion. The date on the right half of the figure indicates that the time is some day in the range of days between the first and last day in the month. τZAMAN supports both intra- and inter-calendar granularity conversions. Additionally, τZAMAN provides classes that model indeterminate times, so the indeterminacy can be accounted for (or discarded if desired) in the conversion.

`April 13, 2003` $\rightarrow$ `April 2003`
(a) `days` to `months`

`April 2003` $\rightarrow$ `April 1, 2003` $\sim$ `April 30, 2003`
(b) `months` to `days`

Figure 6: Granularity conversions

`April 13, 2003` $\leq$ `April 14, 2003` $\rightarrow$ `true`
(a) An "earlier than" predicate

`April 13, 2003` $+$ `5 days` $\rightarrow$ `April 18, 2003`
(b) Adding an interval to an instant

Figure 7: Evaluating temporal operations

The seventh scenario is about supporting arithmetic and comparison operations for time values. Leslie wants to send her banking records to Sydney to be integrated with data from Australian consumers. Leslie observes that Sydney is one day ahead of the USA. To properly integrate the data she needs to convert the data to local conditions in Australia. For the temporal information in her records, she basically needs to add one day to each date. Since her dates are represented in the USA format (`mm/dd/yyyy`), it is more complicated that increasing the "day" number by one; for instance, a day that ends a month would have to increase the month (and possibly the year) and set the day to 1. Increasing a date by one day is just one example of the many arithmetic and comparison operations that applications need to perform on times. An example comparison is illustrated in Figure 7(a), and an example arithmetic operation is depicted in Figure 7(b). The figures show relatively simple operations. In general, these operations can be complicated because the operands may be at different granularities, from different calendars, in different languages, and involve different formats. The times in an operation could also be indeterminate or might even involve special times, such as the variable time called *now* that represents the ever-changing current time [CDI+97]. τZAMAN provides a complete set of temporal comparison operations and a useful set of arithmetic operations. τZAMAN also supports a *semantics* interface that permits users to impose special-purpose semantics for temporal operations, such as converting operands in binary operations to the granularity of the left operand prior to performing the operation.

In sum, many users and applications need temporal functionality. Unfortunately, applications are often limited in their support for time because it is costly to develop the code needed to fully support input and output in a wide range of formats, languages, and calendars, correctly perform granularity conversions, and implement a complete set of temporal operations. What is needed is a flexible, extensible system that supports the modular definition of calendars and granularities, can load new calendars when needed, and can handle all the complexities of parsing and formatting a wide variety of times. The remainder of this paper describes one such system.

3 τZaman Concepts

This section introduces concepts that are of utility to users of τZAMAN, namely *calendars*, *calendric systems* and various *temporal data types*. A calendar is a human abstraction of time. Readers are likely to be most familiar with the Gregorian calendar, but many other calendars are also in daily use. Related calendars are grouped into larger structures called calendric systems. A calendric system facilitates interaction among a

Calendar	*Description*
UTC2	Revised universal coordinated time
Gregorian	Common western solar with months
Lunar	Common eastern lunar
Julian	Western solar with years and days
Meso-american	260 day cycles
Academic	Year consists of semesters
Common Fiscal	Financial year begins at New Year
Academic Fiscal	Financial year starts in Fall
Federal Fiscal	Financial year starts in October
Time card	8 hour days and 5 day weeks
3-shift Work Day	24 hour day divided into three shifts of 8 hours
Carbon-14	Time based on radioactive decay
Geologic	Time based on geologic processes

Table 1: Common calendars

group of calendars. τZAMAN supports temporal operations on three temporal data types: *instants*, *periods*, and *intervals* [JC98]. An instant represents a point on an underlying time-line, a period is the time between two instants, and an interval is a duration of time. In the remainder of this section we explain each concept in more detail. Section 4 presents the τZAMAN architecture to support the concepts.

3.1 Calendars

A *calendar* is a human abstraction of time [JC98]. Calendars define the time values of interest to a user, usually over a specific segment of the physical time-line. A calendar familiar to many is the Gregorian calendar, based on the rotation of the Earth on its axis and its revolution around the Sun. Some western cultures have used the Gregorian calendar since the late 16th century to measure the passage of time. As another example, the Islamic calendar is a lunar calendar, based on the amount of time required for the Moon to revolve around the Earth. Years in the Islamic calendar are counted since the *Hijra* (Mohammed's flight to Medina), which corresponds to the Gregorian calendar year 622 C.E.

The Gregorian and lunar calendars are examples of daily and monthly calendars, but, in general, a calendar can measure time using any well-defined time unit. For example, an employee time card can be regarded as a calendar which measures time in eight-hour increments and is only defined for five days of each week. We note that many different calendars exist, and that no calendar is inherently "better" than another; the value of a particular calendar is wholly determined by the population that uses it. Table 1 lists several prominently-used calendars.

It is important to also support "one-off" or special-purpose calendars. The usage of a calendar depends on the cultural, legal, and even business orientation of the user. For example, businesses generally perform accounting relative to some *fiscal year*. However, the definition of fiscal year varies depending on the business. Universities may have their fiscal calendar coincide with the academic year in order to simplify accounting. Other institutions use the more common half-yearly or quarterly definitions of fiscal year.

To enable calendars to be developed in isolation yet be rapidly integrated into a multi-calendar application, a modular definition of a calendar is essential. The defining characteristics of a calendar can be partitioned into two sets: *intrinsic characteristics* which define the universal qualities of the calendar, and *extrinsic characteristics* which define the user-dependent or varying qualities of the calendar [SS92, Soo93].

Property	*Description*
Locale	Location for timezone displacement
Instant input format	Input format string for instants; there are also formats for now-relative and indeterminate instants.
Instant output format	Output format string for instants; there are also formats for now-relative and indeterminate instants.
Interval input format	Input format string for interval; there is also a format for indeterminate intervals
Interval output format	Output format string for interval; there is also a format for indeterminate intervals
Period input format	Input format string for periods
Period output format	Output format string for periods

Table 2: Calendar properties

The intrinsic characteristics of a calendar define the semantics of the calendar and of its components that depend directly on such semantics. For example, the duration of time units (e.g., week, month) and their interrelationships are intrinsic components of a calendar. Functions performing calendar-defined computations are also intrinsic. An example of such a function would be, `isLeapYear(year)`, for the Gregorian calendar, which returns a Boolean value indicating whether the given year is a leap year.

The intrinsic characteristics of a calendar include a collection of *temporal granularities*. A granularity is a system of measurement for a temporal datum [BDE+98, JC98]. For instance, in the Gregorian calendar, birth dates are typically measured in the granularity of `days` and train schedules are specified to that of `minutes`. Since measurements are discrete, a granularity creates a discrete image of a time-line. More precisely, the underlying time-line can be thought of as being chopped into segments called *granules*. Times are measured to a granule within a granularity.

It is important for a user population to be able to define their own granularities; any fixed system of granularities, such as those supported by SQL from the Gregorian calendar, will not meet the needs of all users. In that sense, a calendar can be defined as a collection of related granularities [WBBJ97, DELS00, BJW00]. Granularities are related in the sense that the granules in one granularity may be further aggregated to form larger granules belonging to a *coarser* granularity [BDE+98]. For example, as every Gregorian year is an aggregation of 365 or 366 days, it follows that `years` is a coarser granularity than `days`. Similarly, `days` is a *finer* granularity than `years`.

The extrinsic characteristics of a calendar capture the *properties* of a calendar that vary depending on the orientation of the user. As an example of this type of characteristic, consider the same date expressed in different languages, say English and Hindi. The Gregorian calendar date may be written as "January/1/1999" in English, but in Hindi it would be "Magha/1/1999". A single date may also be expressed in several formats, e.g., it could be a string like "August 20 2003" or an XML-formatted string such as "<date>August, 20 2003</date>". Both of the formats are in English; however, they are structurally very different. Yet another example is the difference between the `mm/dd/yyyy` format preferred in the United States, and the `dd/mm/yyyy` format used in many other countries. Often, international standards and languages impose a single representation. For example, the ISO 8601 international format represents dates only in the context of the Gregorian calendar and has a rigid set of defined formats [Int00]. In contrast, τZAMAN provides support for user-defined extrinsic characteristics of calendars, and hence can support multiple languages and different formats for dates.

We have identified a set of calendar properties applicable to many calendars. Table 2 lists the properties. Calendars for which a particular property does not apply can ignore the value of the property, if it is defined. Appendix C contains a complete description of the properties in Table 2.

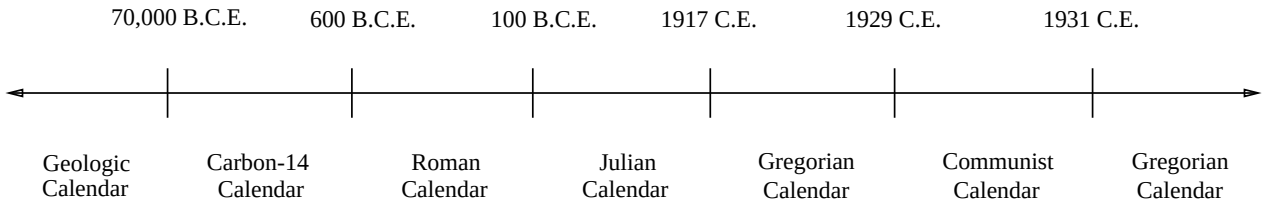

Figure 8: The Russian calendric system

Data Type	*Scenario*
instant	"When did Alice start the race?"
period	"When was Alice running?"
interval	"How long did Alice run?"

Table 3: Examples of temporal data types

3.2 Calendric Systems

Calendric systems are collections of calendars where each calendar covers a contiguous and non-overlapping portion of the time-line, called an *epoch* [JC98]. It is possible that there are times on the time-line that are not covered by any epoch for a calendar in a calendric system. Figure 8 illustrates the Russian calendric system. It captures the use of calendars over time in the area of the world called (in English) "Russia". In the figure, the time-line is not shown to scale. In prehistoric epochs, the Geologic calendar and Carbon-14 dating (another form of a calendar) are used to measure time. During the Roman empire the lunar calendar developed by the Roman republic was used. Pope Julius, in the 1st Century B.C.E., introduced a solar calendar, known as the Julian calendar. This calendar was in use until the 1917 Bolshevik revolution when the Gregorian calendar, first introduced by Pope Gregory XIII in 1572, was adopted. In 1929, the Soviets introduced a continuous schedule work week based on four days of work followed by one day of rest, in an attempt to break tradition with the seven-day week. This new calendar, the Communist calendar, had the failing that only eighty percent of the work force was active on any day, and was abandoned after only two years in favor of the Gregorian calendar, which is still in use today in that country.

τZAMAN is the only system that we know of that supports multiple calendars within a single calendric system. Most systems that support time have only a single, pre-defined calendar over a very small epoch. For example, a DBMS that implements the SQL2 proposal supports only the Gregorian calendar and only over the epoch from 1 C. E. to 9999 C. E. [Dat88, MS93]. This is inadequate for applications that manipulate time values that fall outside of this epoch, such as developing a historical record of ancient Egypt. Also, applications that use time values that are within this epoch, but in a different calendar, cannot be adequately supported. By allowing multiple calendric systems to exist within an application, and supporting calendric systems with multiple calendars, we offer a general notion of expressing time that is able to capture the entire history of an enterprise.

3.3 Temporal Data Types

τZAMAN has three temporal data types with rich semantics that capture the intuitive and familiar concepts of time: *instants*, *periods*, and *intervals*. The data types are explained in detail in the rest of this section; Table 3 gives an example usage for each type.

An *instant* models a single point in time [JC98]. On a continuous time-line, it is generally not possible to precisely identify a single time point because our ability to measure time is inherently imprecise [CR87].

For example, if a wristwatch reports that the current time is 3:45:23 P.M., the time is actually sometime during that second, but it is unknown exactly when. The wristwatch can only measure to the accuracy of the granularity of `seconds`. Usually, an instant is modeled by a single granule. But more generally, an instant is represented by a sequence of granules, called the *support*, together with an optional probability distribution on the support [DS98]. The support indicates the possible granules to which the time is known while the distribution records the probability that the instant is a particular granule. The support extends from a *lower support* granule, l, to an *upper support* granule, u in a granularity, G, and in this paper will be designated using the following notation:

$$l \sim u \equiv \{g \in G \mid l \leq g \leq u\}.$$

It is possible that the lower and upper supports are the same, indicating that the instant is modeled by a single granule. In this case, the instant is called a *determinate* instant. Otherwise, it is called an *indeterminate* instant.

While it is important to recognize that instants are specified only to the precision of a particular granularity, it is equally important to choose the correct granularity. Sometimes, for reasons of linguistic convenience, humans under-specify a time, that is, they specify a time in a very coarse granularity when the time that it signifies is actually known or intended to be at a very fine granularity. For example, if a ship schedule states that a ship departs at 3 P.M., then the time of the ship departure is given in the granularity of `hours`, but "3 P.M." is (probably) accurate to a much finer granularity, specifically to the granularity of `minutes`.

A *period* is a segment of the time-line [JC98]. A period can be represented with a pair of granules. A period that extends from granule g_1 to granule g_2 is the set of granules in G between g_1 and g_2, under the constraint that $g_1 \leq g_2$. Periods literals can be given as either *open* or *closed*; an open period excludes the bounding granule from the period. For example, in the Gregorian calendar the closed period "[1/1/1776 - 12/31/1776]" represents all the days in the year 1776. We will assume that both the starting and terminating granules are in the same granularity. Instants and periods are related in the sense that two instants can uniquely determine a period, and a period's bounding instants can always be determined.

An *interval* is an unanchored duration of time, that is, it is an amount of time with known length but no specific starting or ending instants [JC98]. For example, the interval "one week" is known to have a duration of seven days, but one week can refer to any duration of seven consecutive days. An interval can be either positive, denoting forward motion in time, or negative, denoting backwards motion in time.

It is important to note that intervals do not necessarily have a fixed duration. For example, the length of the interval "one month" in the Gregorian calendar changes from month to month when observed at the granularity of `days`. In February the duration of a month might be 28 days, but in June it becomes 31 days.

Finally, there are some instants that have special semantics. *Beginning* and *forever* are special instants representing the earliest and latest possible times, respectively, that is, minimal and maximal instants. The instant *now* represents the constantly changing current time. A *now-relative* instant includes a displacement from the current time, e.g., *now* + 1 day [CDI+97]. The special instants can be used in periods, and some special intervals also exist. For instance, the interval *all of time* is the duration from *beginning* to *forever*.

τZAMAN supports a basic set of arithmetic operations involving instances of the instant, period, and interval data types. For example, one may wish to determine the arrival time of a train given its departure time and the duration of its trip by adding an interval to an instant, e.g., "March 28, 2003" + "1 day" gives the arrival instant, which is "March 29, 2003". Table 4 shows the supported operations and operands. '`/`', '`*`', and '`+`' are binary operators implementing the operations of division, multiplication, and addition, respectively. '`-`' implements binary subtraction in addition to interval value negation, a unary operation.

Note that the operations are not orthogonal. For example, *instant* * *instant* is undefined since no reasonable semantics for that expression exists.

Operand 1	*Operator*	*Operand 2*	*Yields*
	-	*interval*	*interval*
interval	+	*interval*	*interval*
interval	-	*interval*	*interval*
instant	+	*interval*	*instant*
instant	-	*interval*	*instant*
interval	+	*instant*	*instant*
instant	-	*instant*	*interval*
interval	*	*numeric*	*interval*
numeric	*	*interval*	*interval*
interval	/	*numeric*	*interval*
interval	/	*interval*	*numeric*
interval	+	*period*	*period*
period	+	*interval*	*period*
period	-	*interval*	*period*

Table 4: Valid arithmetic expressions and results

τZAMAN has a complete set of temporal comparison operations. Determining a temporal ordering relationship between a pair of objects is central to many applications. For example, one might be interested in which employees were hired during a particular year, or given two employees, who has more seniority. Allen defined a complete set of relationships between periods [All83]. τZAMAN extends Allen's operators with an analogous set of operators for the instant and interval data types. Table 5 lists the available operations in τZAMAN. This set was shown to be complete elsewhere [SJS95].

The arithmetic and comparison operations discussed above assume that the operands are in the same granularity. In order to have a systematic way of handling operands at different granularities, τZAMAN allows users to define their own *semantics* for operations on temporal data types. Usually this involves converting one operand to the granularity of the other operand. For example, suppose that an interval, say "1 day" known to Gregorian `days` is to be added to an instant, say "12:00, March 1, 2003" at Gregorian `hours`. Below are four reasonable semantics for evaluating the operation.

Mismatch Give a mismatched granularity error [AQdO85].

Left-operand semantics Perform the operation at the granularity of the first operand. This is reminiscent of the assignment operator in many strongly typed languages, which casts the value of the right hand side to the type of the left hand side.

Right-operand semantics Perform the operation at the granularity of the second operand. This is reminiscent of some expressions in C++, e.g., `7/2.0`, which converts the value of the left hand side of the division operator to the floating point type, because the right hand side is a floating point number.

Finer semantics Perform the operation to the finer granularity [CR87, Sar93, WJL91]. If the two granularities are incomparable (neither is finer than the other), then perform the operation to a granularity finer than both arguments; if none exists, give an error.

Coarser semantics Perform the operation to the coarser granularity [BP85, MMCR92]. For incomparable granularities, perform the operation to a granularity that is minimally coarser.

Operand 1	*Operator*	*Operand 2*
interval	`equals`	*interval*
interval	`precedes`	*interval*
instant	`equals`	*instant*
instant	`precedes`	*instant*
instant	`precedes`	*period*
instant	`overlaps`	*period*
period	`precedes`	*instant*
period	`overlaps`	*instant*
period	`precedes`	*period*
period	`equals`	*period*
period	`meets`	*period*
period	`overlaps`	*period*
period	`contains`	*period*

Table 5: A partial list of comparison operators

4 τZaman Architecture

In this section we present the architecture of τZAMAN, and outline how to use the system. The key design features of the architecture are extensibility and service. τZAMAN provides extensibility in two ways. First, it supports multiple calendars, multiple languages, and a wide range of formats for time input and output. Second, τZAMAN can by dynamically reconfigured. Calendars and calendric systems can be dynamically loaded or reloaded with new specifications. τZAMAN provides service by implementing a client/server architecture. A calendar server can be accessed by many remote clients.

We implemented τZAMAN in Java. While the architecture is independent of a particular programming language, the design was influenced by the availability in Java of certain language features. Below we list the six reasons why we chose to implement using Java. First, portability is a big concern. We'd like τZAMAN to operate on most hardware and operating system platforms, even PDAs. The Java Virtual Machine (JVM) provides a stable, platform-independent environment in which τZAMAN can be run. Second, Java is "network-friendly" in the sense that it has strong support for network communication and building client/server systems. We made extensive use of Java's Remote Method Invocation (RMI) classes. τZAMAN can run as a calendar server, providing a network resource for handling times in a specific calendar, such as the Gregorian or Julian calendar. Third, we anticipate that calendar-related data, such as calendar specifications files in XML, will be made accessible on the web. Java classes are available to fetch data using the Hypertext Transfer Protocol (HTTP). Fourth, we anticipate that XML will become popular for representing dates and times. So most of the data that is input and output in τZAMAN, such as temporal constants and calendar definition files, will be formatted in XML. τZAMAN benefits from the widely-used and reliable XML parsing and processing packages of Sun's Java 2 platform, Standard Edition (J2SE) [Mic03]. Fifth, Java supports dynamic class loading. Dynamic class loading can be used to extend a calendar server with new calendars at run-time. Sixth and finally, Java provides support for Unicode. We anticipate that times and dates will be given in a wide variety of character sets.

The remainder of this section presents the architecture for τZAMAN. We first give a broad overview of the major packages and how they are related. Next, τZAMAN is described from a user's perspective. We illustrate how to create a server and client, and how to construct instances of instants, intervals, and periods. Finally, each of the major architectural components is presented in greater detail.

4.1 Overview

Figure 9 shows the major components of the architecture.[2] In the figure, each box represents a group of related packages, each comprised of a number of Java classes, 60 in total. Users are represented with ovals. There are two distinct categories of users: *administrators* and *end-users*. An administrator loads calendars, calendric systems, and language support tables, while an end-user interacts with τZAMAN to manipulate temporal literals. A directed edge in the figure indicates that the source makes use of the methods in some class in the target package. Since τZAMAN is an API the end-user and administrator roles are assumed by a program; in many cases the same program will assume both roles. There are three main operational flows.

1. Configuration — This flow is for configuring τZAMAN, such as loading calendars and properties, and setting up τZAMAN services. Configuration can be performed dynamically, so a configuration flow could happen many times during execution. In Figure 9 the configuration flow is represented by a dashed line.

2. Input/Output — The second flow of operation is related to granularity conversions and input and output of temporal literals. Input calls a temporal data type constructor for an instant, interval, or period. The input and output flow is denoted with a solid line in Figure 9.

3. Operations — The third and final flow is for temporal operations (involving no granularity conversions). τZAMAN provides a set of operations that users can perform on instants, intervals, and periods. The temporal operations flow is denoted with a dotted line in Figure 9.

There are five groups of packages: low-level, calendar-independent aspects (Temporal Data Type and Timestamp), calendar-related aspects (Calendar, Calendric System, Property and Field), the bridge between the calendar-related and calendar-independent aspects (Input/Output), and the system configuration interface (TauZamanSystem and Client/Server).

The Timestamp and Temporal Data Type packages encapsulate the components for the instant, period, and interval data types. The packages are independent of the calendar, although the calendars are used during input (construction) and output of times via the TauZamanSystem and Client/Server packages. A user who wants to create a temporal data type from a string will interact with Temporal Data Type package as shown in Figure 9. For example, an instant can be constructed by converting a string such as "March 14, 2003" to a granule representing the appropriate day in some calendar; possibly it is day 14,562 in the `days` granularity. On output, the instant is converted from a granule to a string by again using a particular calendar and its services. But the temporal data types interact only with the TauZamanSystem and Client/Server packages for input and output as shown in Figure 9.

The Calendric System, Calendar, Property, and Field packages manage access to calendar-related services. The TauZamanSystem and Client/Server packages invoke methods in these packages when users load calendars and calendric systems. Extensibility of calendric systems and calendars is one of τZAMAN's main design features. Calendars can be developed in isolation and then loaded, dynamically, into a running system. Additionally, new formats for input and output of time values can be created and dynamically loaded. The new formats are defined in property specification files. Each new format could have a new language or a new name for a feature in a format (e.g., abbreviated month names).

Figure 9 shows that the Input/Output package bridges the calendar-dependent and independent parts of τZAMAN. When a temporal data type is parsed or formatted, related calendar services are called via TauZamanSystem and Client/Service packages. Input is called when a new instance of a temporal data type is constructed from a string. The string is parsed into individual *fields* using a format specified by a calendar property. The fields are then passed to a calendar, which converts them into one or more granules.

[2] The class structure in JavaDoc can be viewed at `http://www.eecs.wsu.edu/~cdyreson/pub/tauZaman`.

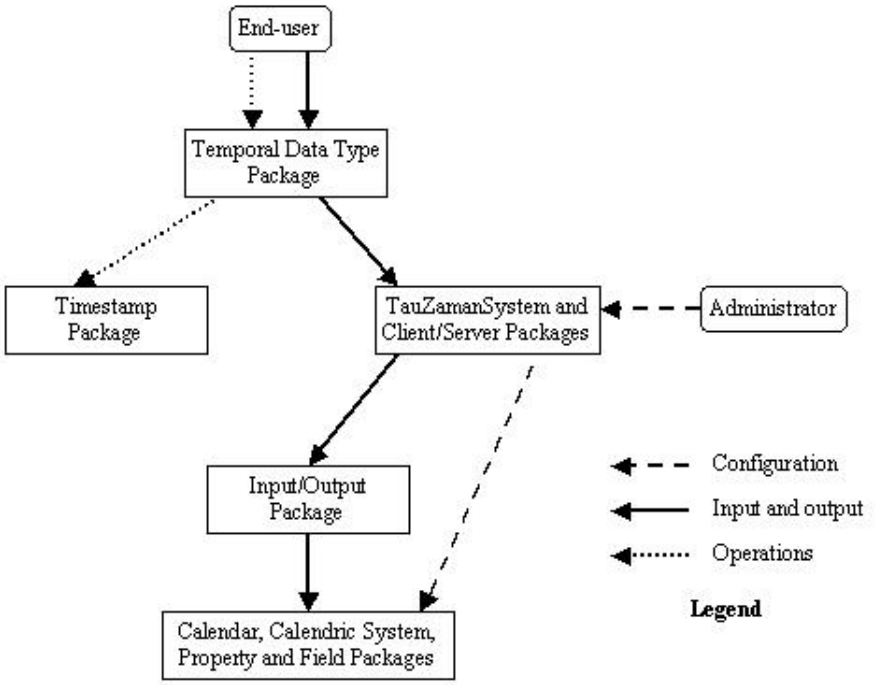

Figure 9: An overview of the τZAMAN system architecture

The granule(s) forms the time in the new instance. For output, the process is reversed. First the granule or granules are converted into individual fields by calling a calendar. Next the string is constructed by using the format specified by an output property.

The TauZamanSystem and Client/Server packages make τZAMAN available to end users. TauZamanSystem is used to perform input and output operations for the Temporal Data Type package. Additionally, a user can use TauZamanSystem to configure calendar-related components, for instance by loading new calendric systems, properties, and calendars.

The Client/Server package lets τZAMAN be run as either a server, a client, or both, as described in more detail in the next section.

4.2 Using τZaman

This section describes each of the flows in more detail, giving code examples of using τZAMAN.

4.2.1 Connecting to τZAMAN

τZAMAN can be run as a server, client, or a single system that is both a client and a server.

Server τZAMAN can be set as a (remote) server. The server provides calendar resources to clients on a network. The server manages all the calendar-related information. Clients communicate with a server using remote procedure call (RPC). A server can support multiple clients. Each client has a separate information space, managed by the server. τZAMAN was designed to minimize the information flow from clients to servers to improve the efficiency of RPC. Typically, each call will pass either a URL, a single granule, or a short list of granules; so the amount of data shipped is small.

Client A client connects to a τZAMAN server over a network. A client can connect to multiple servers. Clients individually manage each server connection as a separate object. A client maintains all instances of the temporal data types, so Instant, Interval, and Period objects reside on the client rather than the server. This means that temporal arithmetic and comparison operations can be performed at the client, without involving the server. The server is involved only in the construction of a temporal data type object, input, output, and granularity conversions.

Local τZAMAN can also be run as a single system that is both a client and server. In this setup, the client and server are on the same machine, and RPC is not used for communication. Only one local service can be run within a process (but the system can still connect as a client to other remote servers).

Finally, we should note that when τZAMAN runs as a server, it can connect as a client to yet other τZAMAN servers, creating a network of τZAMAN servers. So a server that does not know how to handle a temporal literal can pass it off to a server that does.

4.2.2 Running a τZAMAN server

Running τZAMAN as a server is very easy. First, the user must create a `TauZamanSystem` object.

```
TauZamanSystem tzs = new TauZamanSystem();
```

Next, the object is set to be a server.

```
TauZamanSystem.setRemoteService();
```

The server needs to do nothing else; it is now ready to process incoming requests from clients.

4.2.3 Running a τZAMAN client

Making a connection as a client to a server is also straightforward. First a client creates a `TauZamanSystem` object.

```
TauZamanSystem tzs = new TauZamanSystem();
```

Next the connection to the server is established. Below we show the calls to create both a local service and a remote service. The remote service is identified by an IP number. During the creation, the service is requested to load the "UofACalendricSystem" and use the properties (for formatting time values) specified by the "properties.xml" file. Both specifications are XML files; the calendric system specification is given in Appendix B, while the properties are listed in Appendix D.

```
TauZamanRemoteService tzrs = tzs.getRemoteService(
      186.24.12.1,  // IP of the server
      "TauZaman", // Name of service
      "null",     // Use default RPC port
      "UofA",     // Server-side name of calendric system to load initially
      new URL("http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/arizonaCalSys.xml"),
      new URL("http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/properties.xml"));
      );

TauZamanLocalService tzls = tzs.getLocalService(
      "UofA",     // Server-side name of calendric system to load initially
      new URL("http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/arizonaCalSys.xml"),
      new URL("http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/properties.xml"));
      );
```

A problem in making a connection, e.g., a bad URL, will throw a `TauZamanException`. We emphasize that a client will use exactly the same interface for all services provided by a local or remote service; the only distinction is in creating the service. Also observe that the XML specifications need not be local to a server, a server will load each XML file from the HTTP server named in the URL.

4.2.4 Administrator activities

Once the connection to a local or remote server has been established, a client can ask the server to load a calendric system and a default set of properties for that system. Below is an example of an administrator requesting that a local server (`tzls`) load a calendric system.

```
tzls.loadCalendricSystem(
      "UofA clone", // Server-side name of calendric system
      new URL("http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/arizonaCalSys.xml"),
      new URL("http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/properties.xml"));
      );
```

Currently, τZAMAN does not implement levels of security on loading, i.e., anyone can load and name calendric systems. We plan to add security in future.

4.2.5 End-user activities

It is also easy for a client to create and manipulate instants, periods, and intervals. In the design of τZAMAN, we chose to simplify the syntax for creating and manipulating data type instances by adopting the notion of an *active* τZAMAN service and calendric system. Observe that a client could have multiple connections, and each server could have several calendric systems. We let the client establish an implicitly active service and system, to avoid having to specify each in every τZAMAN method. For example, suppose the client has opened a remote service (`tzrs`) and a local service (`tzls`). Furthermore, suppose the local service has two calendric systems: "UofA" and "UofA clone". To make the local service and "UofA" calendric system active in the context of a `TauZamanSystem` object (`tzs`), the user would do the following.

```
tzs.setActiveService(tzls);
tzls.setActiveCalendricSystem("UofA")
```

Once the active service and system have been established, a client can construct and manipulate instances of τZAMAN temporal data types without having to pass τZAMAN specific information to the constructors. Below is an example of calls to the `Instant` and `Interval` constructors.

```
Instant instnt = new Instant("<instant> <year value = "2003"/> </instant>");
Interval intrvl = new Interval("<interval> <year value = "3"/> </interval>");
// An instant is output according to the instant output property
System.out.println(instnt.toString());
```

The permissible format for the XML in each string is specified by the corresponding instant or interval input property in the active service, or the default properties in the active calendric system.

Within a calendric system, the `Instant` object can be converted (cast or scaled) to a new granularity. In the example below, the conversion is from Gregorian years to Gregorian days.

```
Instant dayInstnt = instnt.cast(days); // days is a Granularity object
```

The cast produces a new instant. A cast from years to days will produce the instant corresponding to the first day in the year, i.e., "January 1, 2003". Alternatively, the instant can be cast to a granularity in a different calendar (but within the same calendric system). An example of an intra-calendric system conversion is given below. Assume that the active calendric system has Gregorian and Astronomy calendars.

```
// astronomyDays is a Granularity object in the Astronomy calendar
Instant astroDayInstnt = instnt.cast(astronomyDays);
```

Inter-service system conversions are also supported, indirectly. In the example below, the output of an instant in one calendric system is piped into the `Instant` constructor for the new calendric system (note that an instant caches the calendric system during construction, so changing the active calendric system will not change the semantics of already constructed instants).

```
// Use the local service
tzs.setActiveService(tzls);
Instant instnt = new Instant("<instant> <year value = '2003'/> </instant>");
// Switch to using the remote service, also changing the calendric system
tzs.setActiveService(tzrs);
Instant another = new Instant(instnt.toString());
```

Once an instant, interval, or period has been constructed, it can be compared, added, subtracted, etc. in the context of a `Semantics` object. Left operand semantics casts operands in binary operations to the granularity of the left operand, and then performs the desired operation.

```
Semantics Ops = new LeftOperandSemantics();

// Add the interval to the instant
Instant result = Ops.add(instnt, intrvl);

// Is instnt earlier on the time-line than result?
if (Ops.precedes(instnt,result)) ...
```

The rest of this section provides a discussion of individual components in each group of related packages. We first present the low-level building blocks in τZAMAN, such as components for supporting operations on temporal data types. Next, the high-level components are described, in particular, the TauZamanSystem and Client/Server packages. τZAMAN is implemented in Java; however the design could be implemented using any language or system that supports remote procedure calls, dynamic loading of classes (or functions), and XML parsing and processing.

4.3 Supporting Operations on Temporal Data Types

The classes for the Temporal Data Type and Timestamp packages form the calendar-independent part of τZAMAN. These classes and interactions are shown in Figure 10. In the figure, a solid box represents a class, while a dashed box represents a package. A directed edge indicates that the source class makes use of the target, which can be either another class or a package. Dashed directed edges show the interaction from classes to packages, whereas solid directed edges represent the interaction between classes in the same package. The temporal data type classes use services provided by the TauZamanSystem and Client/Server packages as shown in Figure 9.

The `TimeValue` class is the foundation of the calendar-independent part of τZAMAN. `TimeValue` encapsulates the semantics of the underlying time domain. Many semantics are possible. Time can be modeled as discrete, dense, or continuous; linear or branching; the domain could be bounded or infinite; and time itself could be multidimensional (i.e., a space of valid and transaction time) [JS99]. The `TimeValue` class implements a specific model and provides methods for arithmetic and comparison operations within the model. Only one model can be implemented in a system. We chose to implement a discrete, bounded, one-dimensional time domain. The bounds are the special values, *beginning* and *forever*, representing the earliest and latest possible times, respectively. We used Java's `long` data type for a time, so 2^{64} different times can be represented. In sixty-four bits it is possible to represent current estimates of the lifetime of the universe, approximately thirteen billion years, to the granularity of seconds.

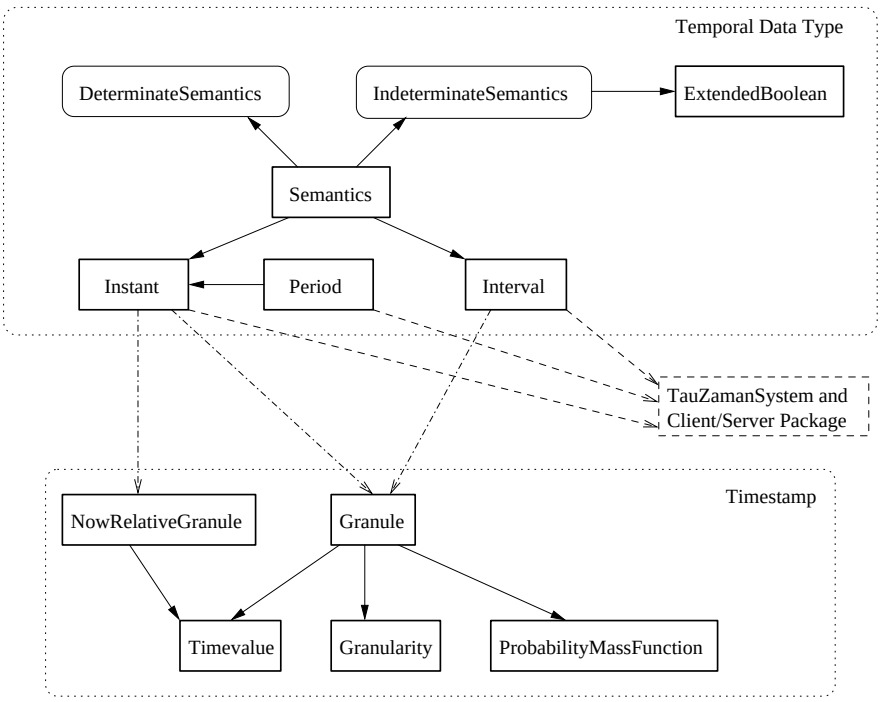

Figure 10: Classes in the Temporal Data Type and Timestamp packages

Each granularity creates a discrete image of the time-line as a sequence of granules. The `Granule` class associates a `TimeValue` object with a granularity to form a granule. For instance, the `TimeValue` with a value of 3 is associated with the granularity of Gregorian `days` to represent the third granule in that granularity. Granules are further classified as determinate, indeterminate, or now-relative. The classification provides additional modeling capabilities. A determinate granule is a single `TimeValue` indicating that the location of the time is known to a single granule in that granularity. An indeterminate granule, however, is a time that is sometime between an lower and upper `TimeValue`, i.e., a set of granules. A `ProbabilityMassFunction` object describes the probability of each indeterminate alternative. Common mass functions, such as uniform and Poisson, can be provided. Finally, a `NowRelativeGranule` instance is a granule that moves with the current time. A now-relative granule may include an interval that displaces the granule a fixed distance from now. Arithmetic and comparison operations are supported for each type of granule.

Granules are part of the data structure in each of the three temporal data type classes: `Instant`, `Period`, and `Interval`. Each of the classes can represent determinate, indeterminate, and now-relative times. String constructors are also provided for each. For example the `Instant` string constructor would create an instant with a determinate granule when given "March 28, 2003", an instant with an indeterminate granule from "March 28, 2003 $\sim$ March 29, 2003", and an instant with a now-relative granule from "now + 5 days".

The arithmetic and comparison operations discussed in Section 3.3 are described by a `Semantics` *interface*. For instance, a `Semantics` provides an operation to add an interval to an instant, but not to add two Instants. `Semantics` is an interface rather than a class because there are several reasonable semantics for performing an operation. For instance one semantics, called *left operand semantics*, converts the right operand to the granularity of the left operand prior to performing the operation. A designer would implement an interface with whatever semantics is desired by the user. The `Semantics` interface is further

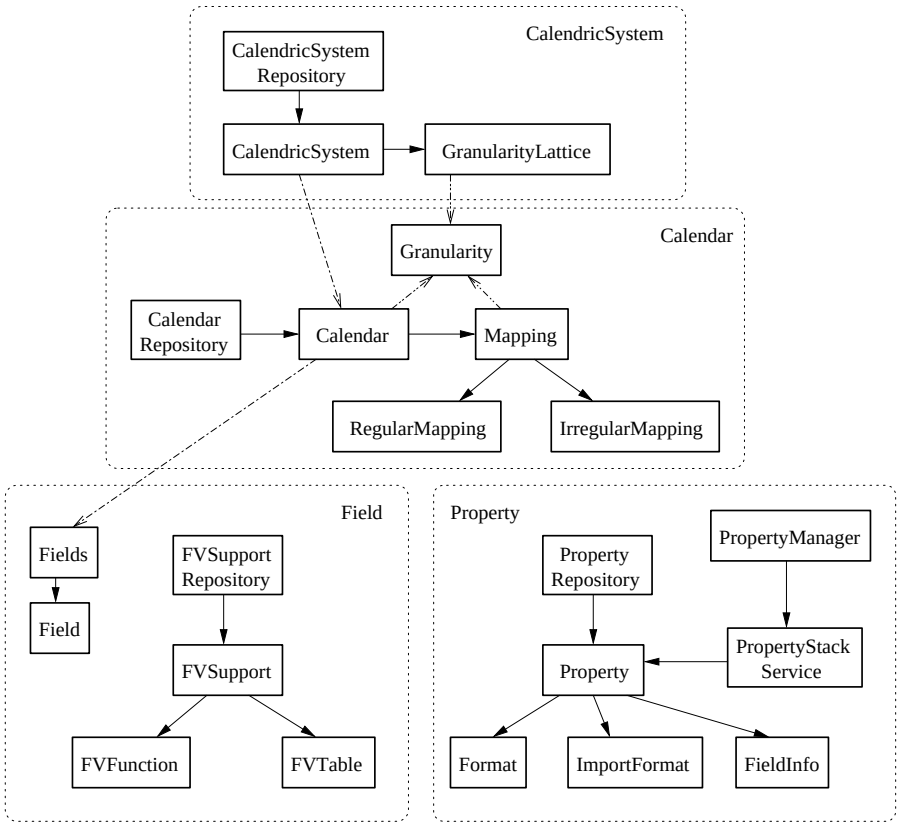

Figure 11: Classes for calendar-dependent components

subclassed into a `DeterminateSemantics` interface and an `IndeterminateSemantics` interface. One difference between the two kinds of semantics is that the determinate semantics returns boolean values for comparison operations, but the indeterminate semantics returns an `ExtendedBoolean` value. `ExtendedBoolean` implements a three-valued logic on the values `true`, `false`, and `maybe`. The indeterminate semantics also permits two controls on the indeterminacy in an operation, called the *plausibility* and *credibility*. These controls are presented in detail elsewhere [DS98].

4.4 Calendar Support

The Calendar, CalendricSystem, Property and Field packages implement the calendar-dependent components of τZAMAN. Users can load, activate and de-activate calendric systems, calendars, properties and field values, convert temporal constants to timestamps, and perform granularity conversions. Figure 11 shows the individual components that comprise the calendar support. Classes in calendar support do not use any other major component of τZAMAN as shown in Figure 9. Solid directed edges represent intra-package interactions, whereas, dashed directed edges represent inter-package interactions between classes.

4.4.1 Calendar

The Calendar package encapsulates a single calendar. We chose to represent an individual calendar as a combination of two different information sets. The first information set consists of the XML specification files for the calendar, granularities, and granularity mappings. Each file is created as part of a calendar development process by a calendar developer. Examples of the specifications for the Gregorian calendar are

given in Appendix A. One of the key features of τZAMAN is that it can dynamically load calendars. It does this by reading the XML specifications for a calendar. So once a developer creates a calendar, it can be made available for loading into a calendar server by simply making the specifications available on the web.

The second information set is the location of Java classes that provide the code to do *irregular* intra-calendar granularity mappings. There are two mapping classes: `RegularMapping` and `IrregularMapping`. Most granularity conversions are *regular* [BDE+98]. A regular mapping can be described completely in the XML specification by a simple formula. For example, the relationship between Gregorian `days` and Gregorian `weeks` is regular since regular periods of seven days group into a week. Code for performing regular mappings is built into τZAMAN. An irregular mapping is a special kind of conversion that is not reducible to a simple formula. One example of an irregular mapping is the relationship between Gregorian `days` and Gregorian `months`. The number of days in a month varies from month to month, and because of leap days the same month may have a different number of days from year to year. Irregular mappings need special code. A calendar developer has to provide a Java class, which is dynamically loaded, to perform an irregular mapping.

With the two information sets, τZAMAN can load everything it needs about a new calendar, provided the calendar specification file is valid. A validating parser can ensure that a specification file is a legal instance that conforms to an XML Schema description of the calendar specification. An exception is thrown if the specification is invalid or other problems are detected during loading.

τZAMAN uses a calendar repository to share calendars among multiple users. The `CalendarRepository` class implements the repository. To prevent duplicate loading of calendars and increase the performance of τZAMAN, when a calendar is loaded it is added to a calendar repository. User requests to subsequently load the same calendar will fetch the already loaded calendar from the repository. However, τZAMAN provides a calendar "refresh" operation to force reloading of a calendar when desired, for instance, if the specification file has been updated. In response to a load request, the calendar repository first determines if the calendar has already been loaded. If found, the repository simply returns the found calendar object, otherwise, it starts to load the calendar from the location identified by the calendar's URL. The URL is used as a primary key in the calendar repository.

4.4.2 Calendric System

The CalendricSystem package implements a calendric system. A calendric system is a collection of multiple calendars. Like calendars, calendric systems are also described by XML specification files. The calendric system specification provides definitions for epochs, calendars, a description of how to integrate multiple calendars, default *properties* (see Section 4.4.3), the location of Java classes to perform irregular inter-calendar mappings, and default regular expressions for date parsing. Appendix B includes an example calendric system specification file, which imports the Gregorian and University of Arizona calendars.

The most important role of a calendric system is to integrate the calendars that it imports. In the calendric system specification each calendar is identified by a URL, which locates the calendar's specification file. The calendars are loaded when the calendric system is loaded. To simplify the writing and handling of calendric system specification files, imported calendars can be given local names, valid within the context of that calendric system. The calendars are integrated by mappings between granularities in different imported calendars. The inter-calendar granularity mappings can be regular, in which case the formula for mapping is given in the specification file, or irregular, in which case the specification file includes a URL to a compiled Java class that performs the mapping. The compiled class is loaded during loading of the calendric system. The calendric system uses the mappings to facilitate granularity conversions [DELS00].

Calendric systems are shared in a repository. To prevent duplicate loading of calendric systems, τZAMAN has a calendric system repository. When a calendric system is initially loaded, it is added to the repository.

Subsequent attempts to load the same calendric system will fetch the already loaded system from the repository. A refresh operation is available to force reloading. The URL of the calendric system specification file is the primary key in the repository.

4.4.3 Property

The `Property` class implements the extrinsic characteristics of each calendar. We identified fourteen kinds of properties, which universally explain user-dependent aspects of a calendar. More specifically, there are properties that define the internal mechanisms of how a temporal literal should be converted to an underlying timestamp. There are also properties that provide other important information, such as a timezone specification, to be used in the input and output of temporal literals. Appendix C summarizes the available properties.

Properties are defined in an XML specification file. A property specification file can contain several properties. A property is identified by the URL of the specification file that defines it and a property name. An example property specification file, containing several individual properties, can be found in Appendix D. A property repository, similar in functionality to the calendar and calendric system repositories, manages property loading and unloading. Having a repository helps to improve the sharing of properties without duplication.

Property values, unlike calendars and calendric systems, are different for each individual application. To support user-specific properties, τZAMAN allocates private *property stacks* to each user. Since the properties have calendar-related components, the stacks are maintained on the server-side, rather than by a τZAMAN client. When a new property is desired, the user asks a τZAMAN server to activate the property. The property is parsed from a specification file (or retrieved from the repository) and pushed onto the stack for that property. Subsequently, users can de-activate the property, causing the property to be popped from the stack.

Properties provide formats for input and output of temporal literals. To illustrate this, assume that a user first wants instants to be parsed according to a "`mm/dd/yyyy`" format. The user would activate a new instant input property with that format. Later the user decides to change the format to "`dd/mm/yyyy`". The user would then activate a different instant input property with the new format. The user could also change other formatting features.

Within τZAMAN the `PropertyManger` class handles the management of properties via the `PropertyStackService` class.

4.4.4 Field

A *field* is an atomic date/time feature of a temporal literal. To illustrate fields, assume we want to construct the instant for the temporal literal "3/20/2003". The literal will be parsed into three fields using the Input instant format property: the *month* field value is 3, the *day* field is 20, and the *year* is 2003. A field generally represents a calendar granularity, but can include other features such as the name of a timezone. As another example, let's assume we want to construct a period from the literal "[March 20, 2003 - March 21, 2003)". The following structure of fields is produced by the parser using the Period input format property: {delimiter="[", {month= '3", day="20", year="2003"}, {month="3", day="21", year="2003"}, delimiter=")"}. This field structure contains two field lists, one for each bounding instant, and two fields for delimiters, which are needed to identify whether the period is *closed* or *open*, on either side.

Fields are also related to language support. When a temporal literal is parsed into fields, each field can be further interpreted by language support tables, called *field value* tables or *fv* tables, that map strings to field values. Consider the literal "Mar/20/2003". After parsing, the month field would have the value "Mar". A field value table would be used to map the string to the value 3 representing the month of March. During

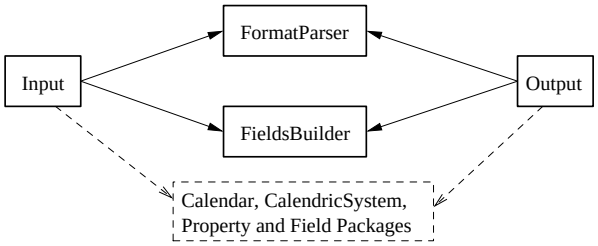

Figure 12: The architecture of the Input/Output package

output the field value tables are used to replace field values (integers) with the appropriate output string. We'll show where these field values come from in the next section.

Field value tables are described in field value specification XML files, which are loaded as part of activating a property. A field value table could be implemented as a Java class. Appendix E gives an example of a field value table specification file. The tables are cached in a repository to facilitate sharing and reuse.

4.5 Input/Output

Input refers to the parsing of a temporal literal during construction of an instance of a temporal data type. *Output* converts the instance to a formatted string. Figure 12 shows the classes and their interactions in the Input/Output package. In addition, it also shows inter-package interactions in parallel with Figure 9. Since output is largely the reverse of input, we will present only the process for input in detail.

Although input can be somewhat complicated due to the possible existence of multiple calendars, languages and a variety of format properties, the input process can be summarized in the following five major steps. The rest of this section explains these steps in detail.

1. Parse the format (an XML file) and build a Document Object Model (DOM).
2. Parse the literal and build a DOM.
3. Match the literal's DOM with the format's DOM.
4. Extract *field values* from the literal's DOM using regular expressions.
5. Create a *field list structure* from the field values.

The first step is to parse the appropriate input property and build a DOM for the format it contains. A format specifies an acceptable skeleton or structure for a temporal literal. Figure 13 shows an example instant input format property with the format enclosed in a `<format>` element. The example format stipulates that only literals consisting of one `<instant>` element with three attributes, `month`, `day`, and `year`, are acceptable. The format further identifies *fields* within the literal to extract for further processing. The presence of a field is indicated by a *field variable*, which starts with a "$" character. There are three variables in the example format: `$month`, `$day` and `$year`.

The second step is to apply the XML parser to the input temporal literal, building a DOM for the literal. Assume that the literal to parse is given below.

```
<instant month="March" day="20" year="2003"/>
```

```
<property name = "InstantInputFormat">
  <value>
    <format>
      <instant month="$month" day="$day" year="$year"/>
      </instant>
    </format>

    <fieldInfo variable="month" name="monthOfYear" using="englishMonthNames"/>
    <fieldInfo variable="day" name="dayOfMonth" using="arabicNumeral"/>
    <fieldInfo variable="year" name="year" using="arabicNumeral"/>
  </value>
</property>
```

Figure 13: An example of an instant input format property

When parsed, the literal will create a DOM with one element node (`<instant>`) and three attribute nodes (`month`, `day`, and `year`). The element has no content (subelements or text). Each of the attribute nodes has a name and a value. As an aside, we note that whitespace within an element tag is not represented in a DOM, for instance there could be one space or five spaces between the `month` and `day` attributes. We further note that the order of the attributes is not recorded in the DOM.

The third step matches the DOM for the literal against each DOM for a format. The DOMs must match exactly, but variables must match at least partially to an attribute value or text value. So variables can only appear where some text is expected. If no format matches, then the literal cannot be parsed by the property. An exception is thrown indicating that the parse failed. The user can try another parse with a new property. If a match succeeds, then the structure of the literal is acceptable, but the variables have yet to be assigned values. In the example given above, the DOM for the literal matches the example format DOM, with the following variable assignments: `$month` = "`March`", `$day` = "`20`", and `$year` = "`2003`". A format can optionally specify whether extra whitespace in text nodes or attribute values is to be ignored during matching.

The fourth step uses regular expressions to extract a value for each variable. The regular expression is built as follows. Each field variable is described by a `<fieldInfo>` element. The field information element identifies a field value table that has all of the possible legal field values. For example the `$month` field uses the `EnglishMonthName` field value table, which is a list of legal month names in English. The table also has a regular expression for recognizing values in the table. For `EnglishMonthName`, the regular expression would specify a non-zero sequence of alphabetic characters from the Western character set. The string "`March`" matches the third entry in this table (see Appendix E), so the value of the `$month` field is 3.

The recognizer regular expression for an `ArabicNumeral` table on the other hand would be a non-zero sequence of digits. The regular expression is applied to the string value that matches the field variable (in the example, each field matches an attribute value). If the expression produces a match, the matched string is checked to ensure that it is in the list of legal values in the table. In this case, there is a Java function that takes the string matched by the regular expression (for `$day`, this string is "`20`") and returns the appropriate integer, 20. The same process generates the integer 2003 for the `$year` field.

The fifth step puts the integers matched by each field variable into a *field list structure* (the `Fields` class). The field list is a collection of information extracted from a temporal literal. A calendar will convert the field list into a granule. For the Gregorian calendar of Appendix A, the field list structure with a `$month` of 3, a `$day` of 20, and a `$year` of 2003, the granule 736004 will be returned (consistent with the origin of the day granularity of January 1, 1 C.E.).

```
<property name = "IndeterminateInstantInputFormat">
  <value>
    <format>
      <indeterminateInstant>
        <support>$lower</support>
        <support>$upper</support>
      </indeterminateInstant>
    </format>
    <importFormat variable="lower" name="InstantInputFormat"/>
    <importFormat variable="upper" name="InstantInputFormat"/>
  </value>
</property>
```

Figure 14: An example of an indeterminate instant input format property

```
<indeterminateInstant>
  <support><instant year="2003" month="March" day="20"/></support>
  <support><instant day="21" month="March" year="2003"/></support>
</indeterminateInstant>
```

Figure 15: An example of an indeterminate instant literal

Indeterminate instants, now-relative instants, and determinate and indeterminate periods all have "bounding" instants. The instant input and output format properties can be imported into the format properties for other temporal data types. Figure 14 shows an example indeterminate instant format property. The upper and lower support are both instants. So the instant input format (e.g., as shown in Figure 13) is used in parsing those components in an indeterminate instant literal, such as the literal shown in Figure 15. The field list structure is slightly more complex for an indeterminate instant; it consists of a pair of lists (one for each support) as illustrated in Figure 16.

4.6 The TauZamanSystem and Client/Server Package

The `TauZamanSystem` class is the manager for access to τZAMAN. Figure 17 shows the class interactions within TauZamanSystem and Client/Server package. In the same figure, interactions with Input/Output, Temporal Data Types and Timestamp packages are also shown.

There were two main design criteria that guided the development of the server/client functionality in τZAMAN.

1. From a client's perspective, there should be no coding difference between using a remote and local service, except identifying the service as local or remote. Our goal was to make the distinction between local and remote objects transparent to a client. However, full transparency can be disconcerting in some distributed system applications since there can be profound differences in performance between using local or remote objects. Therefore, in τZAMAN, a user must simply identify the service as local or remote. Knowing the service type will inform users of potential performance differences.

2. All instances of temporal data types are local. Our goal was to minimize the amount and frequency of client/server communication. Ideally, a τZAMAN client will have all local resources. Remote services will be invoked only when necessary, primarily for input, output, and granularity conversions. Relatively few kinds of objects can be passed from client to server; only the `Granule` class is serializable (and the classes it references, namely `Granularity` and `TimeValue`).

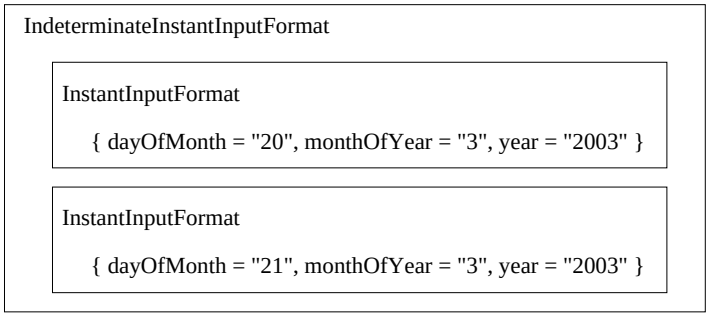

Figure 16: Field list structure for the indeterminate instant

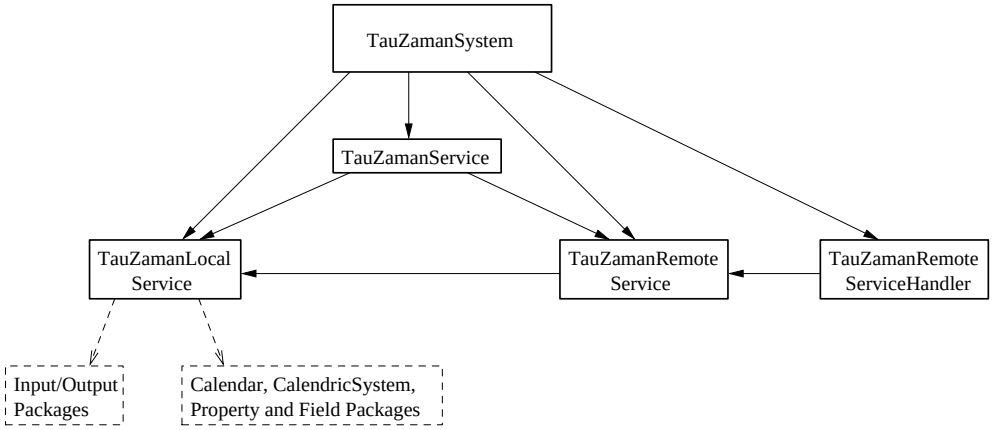

Figure 17: The classes in the TauZamanSystem and Client/Server packages

Figure 18 illustrates the client/server structure. In the figure a `TauZamanSystem` at the client side connects to the server's `TauZamanRemoteServiceHandler`, creating a reference to a `TauZaman-RemoteService` object. The remote service and remote object and depicted with dashed lines at the client side to indicate that the functionality and code physically resides in server side. Note that the `TauZaman-System` object, on the client side, can connect to multiple `TauZamanService` objects. Client/server communication in τZAMAN uses RPC in Java's Remote Method Invocation (RMI) package.

A user, whether it is a client or a server, creates a single `TauZamanSystem` instance. When τZAMAN is run as a server, the `TauZamanSystem` object is responsible for communicating with clients and managing the four repositories. Figure 19 shows the structure of a τZAMAN system and the repositories after a system is created. The repositories are populated over time as a client loads calendric systems, calendars, properties, and field value tables. The choice of setting a system as a client or a server is application-dependent.

A `TauZamanSystem` object also provides `TauZamanService`, which is the client's API for interacting with calendar-related components. A `TauZamanService` offers all of the calendar-related methods to end users. This includes methods to load calendric systems, and calendars, to activate and de-activate properties, and to input and output temporal literals. To increase the flexibility of the system for the users, a user may have several services, connecting the client to a local system and multiple remote servers. The `TauZamanSystem` object stores the currently *active service*; clients switch among the many services by designating the desired service as active.

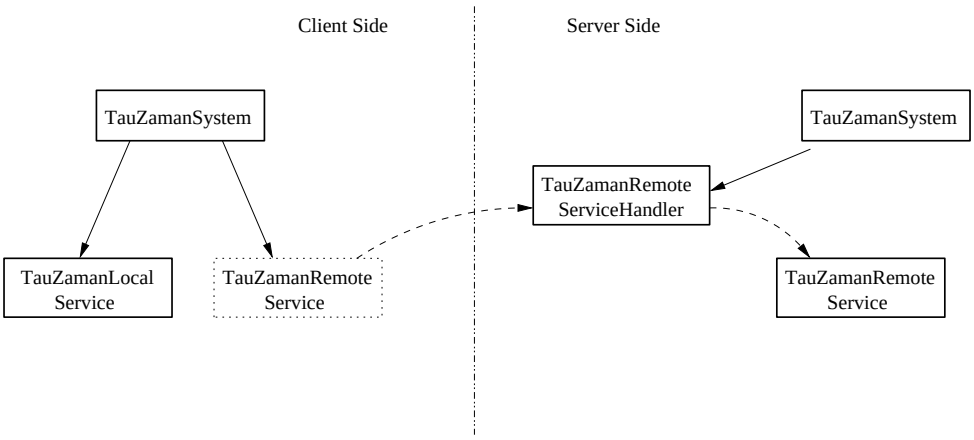

Figure 18: The client/server architecture of τZAMAN

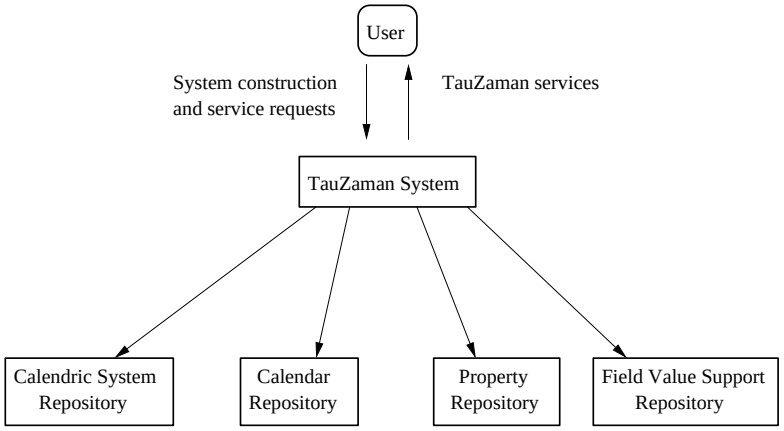

Figure 19: A `TauZamanSystem` after initialization

The `TauZamanService` class is subclassed into two services, `TauZamanLocalService` and `TauZamanRemoteService`. A remote service is designed to be a remote object, that is, it should be registered with the object registry and referenced by a client system. To set up a `TauZamanSystem` as a server, the user should first register the name of the server and publicize the URL of the listener. Any `TauZamanSystem` knowing the URL and registered name can connect to the server as a client.

Each service has an *active state* that stores the current set of active components. The state consists of a calendric system and an operational semantics. A service may have loaded several calendric systems. For example, a client may need the American calendric system, which includes the Astronomy and Gregorian calendars, and in the same service also load the Russian calendric system, which manages the Gregorian and Communist calendars. However, only one calendric system can be active at any time. The client switches between the calendric systems within the service by setting the active calendric system to the desired calendric system. Figure 20 shows a user that is communicating to two services: a `TauZamanLocalService` and a `TauZamanRemoteService`. The local service is the currently active service. It includes two calendric systems: Russian and American. The American calendric system is currently active.

Maintaining a "global" active service and active state within that service reduces the overhead on temporal data type operations. For instance, consider an Instant constructor. Instead of having to pass the active service and active calendric system to the constructor, the active service and state are retrieved within the

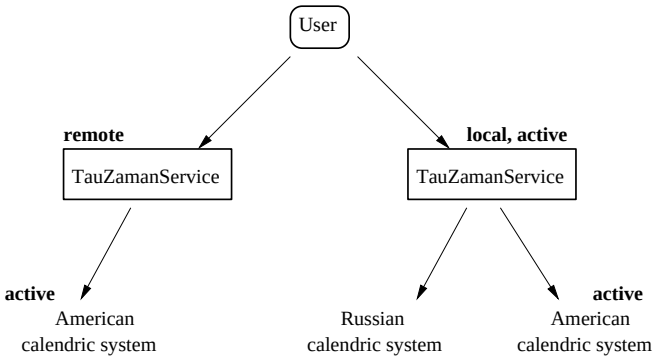

Figure 20: Communicating with multiple services and calendric systems

constructor using class methods in the `TauZamanSystem` class. This minimizes the length of parameter lists in methods. Furthermore, two of our design goals were to support client/server services and be able to utilize multiple calendric systems. Temporal data type operations in τZAMAN, even with the additional functionality, need only a reasonable number of parameters, e.g., the operation to add an interval to an instant is invoked with only the interval and the instant. In τZAMAN, the active service is cached in each created instance of a temporal data type, along with the active state of the service so that the instance can be later output using the same service and calendric system. If a cached service or calendric system is subsequently unloaded or dereferenced then an exception will be thrown when operations on those temporal data types are invoked.

5 Coding Statistics and Experimental Results

This section reports on the size and performance of τZAMAN. Statistics about the τZAMAN's implementation are given first, followed by results of several performance experiments. We measured the performance of local and remote τZAMAN services in configuring a system and input and output of temporal literals.

5.1 Coding Statistics

The current version of τZAMAN consists of approximately 12,500 lines of Java code, not including the code in system or third-party supplied classes. τZAMAN has 60 classes organized into 8 packages. We developed τZAMAN using Sun's j2sdk1.4.1_02 environment. We did not attempt to optimize performance with a native-code Java compiler, or by tuning the code with a Java profiler. τZAMAN has several package dependencies. The dependencies are listed below along with the tasks for which each is needed.

- `java.rmi` is used to implement RPC behavior.
- `java.net.URLClassLoader` is used for dynamic loading of methods and classes for irregular mappings.
- `javax.xml.parser` and `org.w3c.dom` are used to parse and process the XML-formatted specification files, and during input and output of temporal literals.
- `java.util.regex` is used for to match regular expressions for field values during parsing of temporal literals.
- `java.util.Hashtable` is used extensively for implementing the repositories.

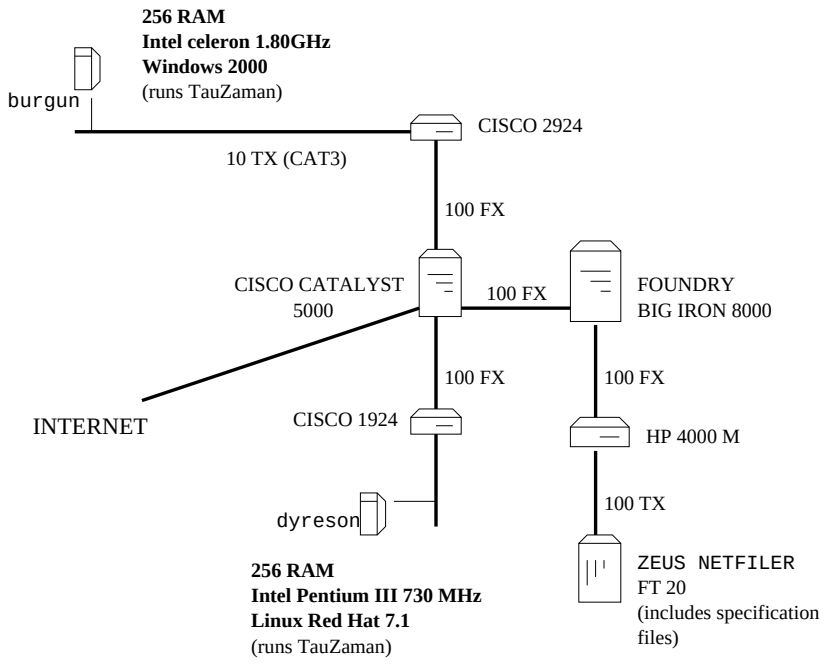

Figure 21: A diagram of the environment for the experiments

5.2 Experiment Environment

We conducted the experiments in a distributed system environment because τZAMAN is a client/server system. Figure 21 shows the network architecture for the machines in the experiment. The two primary machines in the environment are `burgun` and `dyreson`. `burgun` is a Windows box, while `dyreson` runs Linux. We measured the round trip time between `burgun` and `dyreson` at approximately eight milliseconds for a dummy Java RMI call. Both machines are served by a network file server called `zeus`, so loading and unloading involve fetching files from `zeus`. We used Java 2 SDK, version 1.4.1.

5.3 Experiments on TauZamanService Initialization

A client accesses τZAMAN by constructing a `TauZamanService` object, which could be remote or local. The service is started by providing the URL of a calendric system specification and a property specification. The specification files are fetched (via HTTP), parsed, and processed to form the default components of the service. In processing the calendric system specification, further fetches are done for each calendar managed by the calendric system. Starting a service also initializes the repositories.

The first experiment measures the performance of creating a local service. We used the specification files given in Appendix B and Appendix D. We started a local service on `burgun` by providing the URLs of a calendric system and property specification located on `zeus`. We subsequently recreated the same local service to test the performance of reloading the system (with objects cached in the repository). Table 6 gives the measured times. All times are rounded up to the millisecond. All times in this and subsequent figures represent a single round trip, unless noted otherwise. Since the service is local, there is no network cost on the creation, or recreation of the service. The initial loading time is, not surprisingly, much longer than subsequent loading times because τZAMAN provides repositories to cache reused objects. Note that the initial loading time is a "one-time" cost.

	Total	Calendric System	Property Table
Initial loading	633	518	115
Subsequent loads	1	1	1

Table 6: Average loading times (in milliseconds) of a `TauZamanLocalService`

	Client Side	Server Side		
	Total	Total	Calendric System	Property Table
Initial loading	720	686	499	187
Subsequent loads	30	1	1	1

Table 7: Average loading times (in milliseconds) of a `TauZamanRemoteService`

The second experiment measures the performance of creating a remote service. In this experiment the client is located at `dyreson`. The client creates a remote `TauZamanRemoteService`, identifying `burgun` as the remote server. `burgun` loads the calendric system and property table specification files from `zeus` via HTTP in response to the request. The results are given in Table 7. We averaged the times over five tests to smooth the effects of network congestion, which lead to variations of up to 40 milliseconds per round-trip. We separated the total time (client side) from the load time (server side). As with the local service performance, the time of the initial load is longer than subsequent loads due to caching in the repositories. Note also that the initial load time on the client side time is longer than that for the local service. The reason is that there is overhead on establishing communication between the client and the remote server that is only incurred with a remote service. In addition to the overhead of network, round-trip time there is an additional cost because a `TauZamanRemoteService` object is marshaled and unmarshaled during the call. As stated previously, this is one of the reasons that we did not pursue a fully transparent distributed architecture, since response times are longer with remote services.

The third experiment tests the performance of input and output of temporal literals. The operations perform effectively the same amount of work, just in a different sequence. So we will measure the total cost of performing an input followed immediately by an output. The experiment tests six different kinds of temporal literal: determinate instant, now-relative instant, indeterminate instant, (determinate) interval, determinate period, and indeterminate period. period, and interval, and their indeterminate and now-relative formats. Appendix D gives the format properties used in the experiments for each kind of temporal literal. These formats are of normal complexity rather than worst-case complexity. More complex formats will incur a slightly higher cost. The literals tested are given in Appendix F.

We first experiment on a local service; the next experiment is for a remote service. The local test is performed on both `burgun` (a Windows box) and `dyreson` (a Linux box). `burgun` has more memory and a faster CPU. A `TauZamanLocalService` is created on each machine, with specification files fetched from `zeus`. Table 8 reports the results of the experiment. Like the other experiments it is separated into two different measurements: an initial loading (for the first input and output) and a consecutive loading time. The times are given in milliseconds. The initial cost is higher than subsequent I/O because initially the format is parsed and the field value tables are fetched from `zeus`; on subsequent conversions, the parsed format and field value tables are retrieved from a repository.

The conversion times differ for the different kinds of literals. The indeterminate period is the slowest, while the determinate interval is the fastest. The differences in the timings are because an indeterminate period is composed of four instants, so we would expect it to take a bit longer than I/O of a single instant. The determinate interval is the fastest because it has the simplest format. `burgun` performs better than `dyreson` due to better hardware on `burgun`.

	dyreson		burgun	
	Initial I/O	Subsequent I/O	Initial I/O	Subsequent I/O
Instant	199	8	130	7
Now-relative Instant	226	12	150	8
Indeterminate Instant	249	17	155	13
Period	278	14	175	10
Indeterminate Period	330	19	220	14
Interval	159	6	100	5

Table 8: Average input and output times (in milliseconds) for different kinds of temporal literals using a local service

	dyreson	
	Initial I/O	Subsequent I/O
Instant	287	29
Now-relative Instant	383	41
Indeterminate Instant	390	53
Period	365	35
Indeterminate Period	402	37
Interval	258	28

Table 9: Average input and output times (in milliseconds) for different kinds of temporal literals using a remote service

We next tested input and output in a remote service. We used exactly the same experiment as for the local service, except that we used a remote service from a client on `dyreson` to a server on `burgun`. This test includes the overhead on the network communication and marshaling of parameters, so the overall cost should be greater than that of the local service.

Table 9 shows the results of this experiment. The cost of the initial loading includes the time spent fetching field value tables. Consecutive I/O costs are much lower. When compared to the local service test, we can observe the overhead in the remote communication. The times in Table 8 are lower than those in Table 9. The last observation to make about the results is that now-relative and indeterminate instants are even more expensive. The reason is that there is a single Instant constructor, rather than separate constructors for determinate, now-relative, and indeterminate instants. During construction of an instant, the (determinate) InstantInputFormat property is used to parse the instant. If the parse fails then the the NowRelativeInstantIntputFormat is tried, followed by the IndeterminateInstantInputFormat. Each parse failure results in another round of RPC between the client and the server until the appropriate kind of instant is finally constructed. (We could have had the server try each kind of property in succession. This would improve the times slightly, since it saves on one or two network round-trips. But overall the cost is dominated by the parsing, so such a refinement wouldn't have a large impact on the performance.)

6 The TAUZAMANTESTER- A Graphical User Interface (GUI) for I/O

τZAMAN provides both an Application Programming Interface (API) for programmers and a (prototype) GUI-based testing tool called the TAUZAMANTESTER. The TAUZAMANTESTER is a user-friendly tool for two user groups: application developers who want to see τZAMAN in action before writing code, and

specification developers who want to test and debug their specifications. More specifically, the tool was designed to meet the following goals.

- To provide a nice interface for demonstrating τZAMAN.
- To create a platform for testing the input and output of temporal literals. Rapid testing can decrease the time needed to develop format properties and other XML specifications. The testing includes checking the specifications for syntactic correctness and completeness.
- To simplify τZAMAN's configuration and setup for a naive user. Users can activate a service by selecting it from a combo-box instead of by writing code. Different service configurations can be loaded using the tool.
- To facilitate the testing of temporal operations. Users can compare or perform arithmetic on created time values.
- To provide performance measures. The tool reports the processing time of each operation.

When the TAUZAMANTESTER is started, a main window is opened. A screen shot of the main window is shown in Figure 22. The window includes a top row of four 'tab' buttons. Choosing a button puts the TAUZAMANTESTER into one of the following four modes.

1. Service Configuration - A user would select this tab to test and load new configurations for a local or remote service. Successfully loaded configurations become available in the other modes.
2. Property Management - Allows a user to test property specifications and load new properties (for testing in the other modes).
3. Input/Output - Converts a time literal, as described in more detail below.
4. Temporal Operations - Provides an interface for performing arithmetic and comparison of temporal literals.

When the 'Input/Output' mode is selected, the TAUZAMANTESTER creates an instance of `TauZamanLocalService` and `TauZamanRemoteService`. All of the configuration information, such as the URLs of the calendric system, property table, and calendar specification files, as well as the URL of the remote τZAMAN server are specified in an XML-formatted initialization file for the tool. The configuration can be changed easily. We provide a default calendric system specification file listed in Appendix B, a default Gregorian calendar given in Appendix A, a default Academic calendar (not given in this paper), and a default list of properties given in Appendix G. After initializing the services, the TAUZAMANTESTER opens the main window and waits for user instructions. In 'Input/Output' mode, there are three panels.

1. An input panel has a scrollable text area in which user can input a temporal literal. The input panel is the left-most area in the screen shot in Figure 22.
2. A configuration panel provides a set of GUI components that allow the user to configure the `TauZamanSystem` object. The configuration panel is the list of buttons and combo-boxes in the middle of the screen in Figure 22. For example, a user can activate the remote or local τZAMAN service by selecting the appropriate one from a combo-box. Or a user could choose from among several properties to active one for input and output formatting. The user must select one of the temporal data types to parse and output the time literal entered in the input panel.

3. An output panel has a scrollable text area that displays the output of the temporal literal. Note that the input and output use different formats, as listed in Appendix G. The output panel is the right-most area in the screen shot in Figure 22.

Generally a user will enter a temporal literal into the input panel, configure the settings as desired, press the "PROCESS" button in the configuration panel, and the output will appear in the output panel. If any exceptions occurs during processing, for instance the input did not parse correctly, an error dialog box will pop up. The GUI also displays the processing time of the entire operation.

Two snapshots of the TAUZAMANTESTER in action are shown in Figure 22 and Figure 23. In Figure 22, an indeterminate instant is entered in the input sub-panel, TauZamanSystem is configured using the components in the configuration sub-panel, and the instant is output. In Figure 23, a determinate period is processed.

The current tool is limited to input and output; we plan to extend the tool to showcase τZAMAN's other capabilities in the future. Some of the important future functions we plan to implement within the tool include the ability to

- dynamically configure the services, such as adding a new service, removing an existing service or setting its `TauZamanSystem` object to be a server;
- improve the visibility of the property management by showing the property stack; and
- add a full set of arithmetic, comparison, and granularity conversion operations.

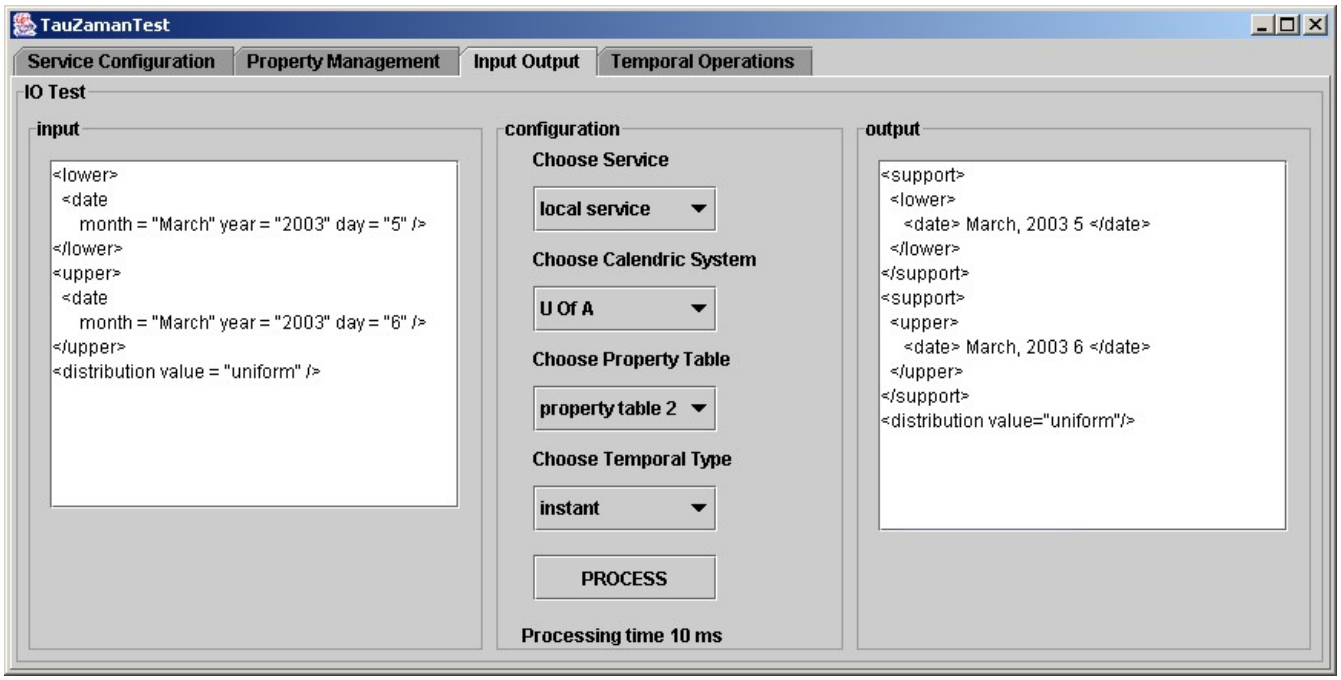

Figure 22: A snapshot of the TAUZAMANTESTER processing an indeterminate instant

7 Related Work

While work on temporal data types goes back two decades, in the last five years there has been a flurry of activity. Related research can be broadly classified into two categories: modeling and implementation.

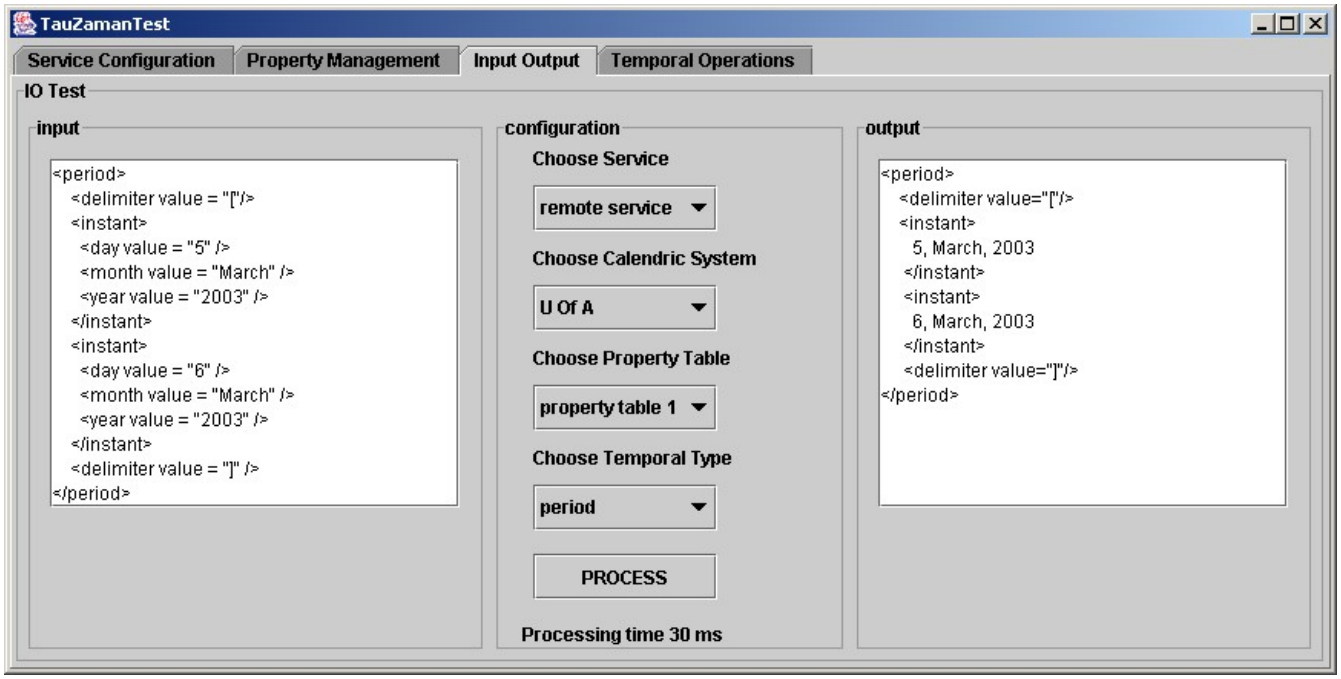

Figure 23: A snapshot of TAUZAMANTESTER processing a determinate period

The modeling category covers research in temporal data models, and in particular, it establishes desirable operations on temporal data and calendars. The second category is research into implementations of the first category. Although this paper is in the implementation category, in this section we trace the influences on this research from both categories.

Allen motivated the *interval* (which we call a *period*) as a fundamental temporal entity [All83]. He formalized the set of possible relationships which could hold between two intervals and developed an inference algorithm to maintain the set of temporal relationships between entities.

Anderson described a formal framework to support conceptual time spaces using inheritance hierarchies [And82, And83]. Her model also supports multiple conceptual times. τZAMAN can be considered as a practical realization and extension of some of the concepts developed by Anderson.

Clifford and Rao developed a framework for describing temporal domains using naive set theory and algebra [CR87]. Their framework allows a hierarchy of calendar independent domains to be built and temporal operators to be defined between objects of a single domain and between objects of different domains. The framework is powerful but lacks the ability to describe time domains that are not integral multiples of finer granularity time domains. For example, `months` are not an integral number of `weeks` since a whole number of weeks do not ordinarily correspond to a single month. Our work removes this limitation by making the semantics of any conceptual time unit user-definable. The user is not tied to any predefined notion of time or time domain.

Navrat and Bielikova argued for declarative, rather than algorithmic, calendar definitions [NB95]. Algorithmic definitions sometimes lead to oversimplification of predictions for future times and unnecessary approximations of past times. For example, in the Islamic calendar, the first day of the month of *Ramadan* cannot be predicted by an algorithm, although an approximation exists and is used by some cultures. Navrat and Bielikova addressed this problem by using factual past knowledge combined with Prolog to better define the start of Ramadan. Their framework also provides some support for multiple calendars, and inter-calendar calculations. But accounting for the semantics of granularity in operation is missing.

Bettini, Wang, and Jajodia developed a formal foundation for reasoning about temporal granularities [BJW00]. Ning, Wang and Jajodia extended this work with an algebraic approach to define granularities and calendars [NWJ02]. They argued that irregular granularity conversions can be done in a declarative way without the need of a specialized piece of code, although the declarative specification can be complicated. In contrast, τZAMAN uses specialized code. We are investigating using their approach to support irregular mappings in τZAMAN.

Kraus, Sagiv and Subrahmanian proposed a formal definition of calendars and temporal data types in terms of constraints, as opposed to our representation, which are granules (as integers) at different granularities [KSS96]. They also showed how to support multiple calendars and argued that specifying a time point as integers or real numbers is cumbersome for human beings. We agree: τZAMAN uses familiar string representations for temporal literals.

Kakoudakis and Theodoulidis implemented a single calendar system that supports operations in only the Gregorian calendar, with a limited number of granularities [KT96].

Chandra, Segev and Stonebraker [CSS94] presented a design for set-based specification of calendars. They gave an algorithm for parsing the specifications and described how to extend the temporal support in the Postgres database management system with new calendars. Chandra et. al compared their project to MULTICAL [SSD+92], which is a precursor project to τZAMAN (as described in more detail below).

In the second category, implemented systems, there exist several systems that support temporal data types, temporal operations, datetime calculations and conversions. Most of the systems that perform datetime calculations and conversions are limited in scope, having static calendar support, with at most four or five different calendars, and limited kinds of formats. On the other hand, there are also applications that support multiple calendars and temporal data type operations.

τZAMAN is an enhancement of two earlier systems: MULTICAL [SSD+92] and TIMEADT [KLS99]. MULTICAL adds support for time and multiple calendars to relational database management systems. MULTICAL has a core system of calendar-independent temporal operations, but allows users to modularly define calendars for formatting times in different calendars and languages. MULTICAL does not have a predefined set of calendars; rather new calendars can be defined and compiled into the system. TIMEADT is a successor to MULTICAL. It refines the temporal operations in MULTICAL by adding support for granularity and temporal indeterminacy and support for C++. τZAMAN inherits many design features from both MULTICAL and TIMEADT, but is different because it can dynamically load calendars, can parse temporal literals formatted in XML, provides a client/server system, supports calendars, etc., as XML documents accessible on the Internet, and is implemented in Java.

Boost [Sof02] is a date-time library in C++, which supports three basic temporal data types: point, duration and interval. One of its design main goals is to support ISO 8601 compliant input and output representation. It provides arithmetic and comparison operations for each type, although not with different semantics for granularity conversions. It also has iterators on time and date ranges, which helps a user iterate over the days of a week, for example. Boost supports multiple calendars, but not the dynamic loading of calendars. It also lacks inter-calendar conversions and calculations.

International Components for Unicode (ICU) [Cor02] is a set of libraries developed by the Unicode group in IBM Globalization Center of Competency. The main goal of these libraries is to hide the cultural and geographical differences in international software development. One of the problems that ICU addresses is the multi-cultural aspect of representing time by supporting multiple calendars and timezones. Currently ICU only supports the Gregorian calendar but with its abstract calendar structure it is claimed to handle multiple calendars, again in a static context as in Boost. Additionally it supports only a limited number of granularities, and does not handle inter-calendar conversions.

The Joda project includes a re-implementation of Sun Java's built-in date and calendar classes [Col02]. Its main aim is to provide date and time implementation to the Java community. Joda supports multiple

calendars, but not dynamic loading of calendars. It provides ISO 8601 compatible input and output. Additionally it provides an API that includes methods to create input and output formats. The format can be checking for correctness before it is tried in input or output. In our project formats can be produced in XML files and thus can be shared and examined easily. In Joda, to relate an instance of a temporal data type, like an instant, to a calendar, a user has to pass the calendar object to the instant constructor. τZAMAN globally caches the active TauZamanService and calendric system to keep the parameter lists short. Joda supports period and interval temporal data types under the name of TimePeriod, which we believe confuses this distinction. Joda argues the immutability of temporal data types for being safe in thread environments; TauZaman also has immutable temporal data types. And lastly Joda does not include temporal conversions between different temporal data types.

WebCal [Ohl03b], a calendar server produced by OhlBach, is a client/server architecture for providing temporal support. WebCal is a part of the WebTNSS [Ohl03a] project, which is a support system for temporal notions. WebCal provides calendar-independent time representations and temporal operational support in WebTNSS. Although τZAMAN and WebCal are similar in that they are both client/server systems, there are also several differences.

- τZAMAN's specification files simplify the production, understanding and publishing of calendars, calendric systems, properties, and field value tables. An application that uses WebCal must code these features into the application.
- In WebCal the smallest granularity is `seconds`. τZAMAN can support much finer granularities.
- In WebCal, all temporal operations are built on top of an interval data type. τZAMAN differentiates between instant, interval, and period data types. τZAMAN also supports indeterminacy and now-relative values.
- For performance reasons, τZAMAN can be setup to run in a single process with a local service, but WebCal is only a server.
- WebCal does have language support, partly because it is designed to provide calendar-independent time representation for the WebTNSS project. On the other hand, τZAMAN supports language-dependent formats in time values.

GSTP [Bet03] is a client/server system that provides granularity conversions and multi-granularity constraint satisfaction. τZAMAN provides the former (as well as many other services) but not the latter. GSTP only runs as a server; τZAMAN can be set up as a server, a client, or both.

Finally, there are several papers devoted strictly to the parsing of dates.

Karttunen et al. [KCGS96] proposed a regular expression calculus for natural language applications. And as one of its illustrations, they described a finite state grammar for dates. For a completely new date input, a new grammar should be employed. On the other hand, in our approach users can create a format by writing an XML fragment and dynamically add it into the system to handle a new date input. But obviously this context is application-specific and the extent of Kartunnen et al.'s proposal is very broad.

Sperberg-McQueen [SM99] argued that recognition of dates is possible by regular expressions and gave `lex` code that recognizes and validates ISO 8601 complaint dates.

Cameron [Cam99] provided a set of shallow parsing regular expressions, which can be used to parse an XML document into individual items, such as attribute and text values. He argued that this style of parsing is relatively faster than off-the-shelf XML parsing and processing tools. We chose to implement a different approach in τZAMAN. τZAMAN uses an off-the-shelf XML parser to match the XML skeleton in a format against that of a literal, and located the text and attribute values. Parsing in τZAMAN then isolates the fields

within text nodes or attribute values with regular expressions. One problem with using regular expressions for an entire XML fragment is that they can be very complex and hard to understand when combined with the regular expressions fetched from the field value tables for individual fields. We chose to make our format properties easy to specify.

8 Conclusions and Future Work

τZAMAN is a system written in Java for formatting and manipulating times and dates in multiple calendars and languages. τZAMAN has a dynamic and extensible architecture that separates calendar-dependent from calendar-independent aspects of processing time values. From a design perspective, τZAMAN redesigns and extends all of the basic mechanisms previously employed. From an implementation perspective, τZAMAN achieves full dynamic support for calendars and related components in a client/server system, and brings a new, XML-based information representation and processing style. The primary contributions include the following.

- τZAMAN supports dynamic and distributed handling of calendars and other services. We take advantage of Java's dynamic class loaders to provide dynamic support for extending servers on the fly with new calendars and calendar-related components. τZAMAN implements a client/server model that makes calendar-related services available on a network.

- XML technology is used to represent and process critical data. τZAMAN improves the representation and processing of the system specification files by formatting the files in the XML. This also improves the processing of the specification files and allows them to be shared on the web. Finally, τZAMAN integrates XML into the parsing and output of temporal literals, to meet the future growth of times formatted in XML.

- Repositories allow effective sharing of components. τZAMAN uses repositories to enable sharing of critical data, such as calendars, calendric systems, granularity mappings, formats (as properties) and languages (as field values) for parsing temporal literals. Repositories reduce the response time for users, especially when parsing temporal literals and performing granularity conversions, by caching components that are reused.

In future we plan to extend this research in several directions. The first direction is improving the speed of τZAMAN. There are several optimization techniques that could be implemented. One optimization is to batch input, output, and granularity conversion requests to remote servers to amortize turnaround time. A second optimization is to cache granularity mappings on the client side, to avoid an RPC call to cast or scale an instance of a temporal data type. A third optimization is to skip the expensive, useless step of parsing a non-XML temporal literal with an XML parser.

Another direction of future work is studying how to craft user interfaces to ease the task of calendar specification and reduce the possibility of mistakes by calendar developers. This would involve extending the TAUZAMANTESTER with support for calendar and calendric system debugging and configuration. The tool could also be extended to visualize granularities enabling developers to graphically construct granularity mappings, to create and use probability mass functions for indeterminate temporal data types, and to have a point-and-click interface for creating input and output formats.

A third direction for future work is to refine and extend the mappings between granularities to include granularities with "holes," e.g., there are some days that are missing between granules in `holidays`. In this context, detailed experiments on temporal operations and conversions between different granularities will be performed. A fourth direction is to integrate τZAMAN with Xalan [Pro03]. Xalan is an XPath evaluation

engine. The idea is to engineer Xalan to coordinate with a calendar server to provide "temporal views" of XML fragments that correspond to time literals in an XML document. So, given an XML document that has time literals in ISO format, a user could query the document using a view of those times in any desired calendar and format, for instance, in the Islamic calendar. A fifth direction is to describe the client API as a web service. This would allow web bots and shopping agents to make direct use of τZAMAN's functionality. A sixth direction is to add "pull" technology for calendar, property, language support, and calendreic system specifications. Currently, when a specification changes, the specification has to be manually reloaded into a running τZAMAN server. By automatically reloading such files when they are modified, calendars and other components can be kept up-to-date. We'd also like to optimize the performance of τZAMAN by using a native-code Java compiler and tuning the code with a Java profiler. Finally, we'd like to re-implement Java's current calendar support in τZAMAN. This will help to demonstrate the extensibility of τZAMAN.

Acknowledgments

This research was supported in part by NSF grants IIS-0100436 and EIA-0080123.

References

[All83] J .F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the Association of Computing Machinery*, 26(11):832–843, November 1983.

[And82] T. L. Anderson. Modeling Time at the Conceptual Level. In P. Scheuermann, editor, *Proceedings of the International Conference on Databases: Improving Usability and Responsiveness*, pages 273–297, Jerusalem, Israel, June 1982. Academic Press.

[And83] T. L. Anderson. Modeling Events and Processes at the Conceptual Level. In S.M. Deen and P. Hammersley, editors, *Proceedings of the Second International Conference on Databases*, Cambridge, Great Britain, 1983. The British Computer Society, Wiley Heyden Ltd.

[AQdO85] M. Adiba, N. B. Quang, and J. P. de Oliveira. Time Concept in Generalized Data Bases. In *Proceedings of the ACM Annual Conference*, pages 214–223. Association for Computing Machinery, October 1985.

[BDE+98] C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass, and X. S. Wang. A Glossary of Time Granularity Concepts. In *Temporal Databases: Research and Practice, Lecture Notes in Computer Science 1399*, pages 406–411. Springer-Verlag, 1998.

[Bet03] C. Bettini. Web services for time granularity reasoning. In *Proceedings of the International Symposium on Temporal Representation and Reasoning*. IEEE, 2003.

[BJW00] C. Bettini, S. Jajodia, and X. S. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning*. Springer-Verlag, San Mateo, CA, 2000.

[BP85] F. Barbic and B. Pernici. Time Modeling in Office Information Systems. In S. Navathe, editor, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 51–62, Austin, TX, May 1985. Association for Computing Machinery.

[Cam99] R. D. Cameron. REX: XML Shallow Parsing with Regular Expressions. *Markup Languages*, 1(3):61–88, 1999.

[CDI+97] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of "Now" in Databases. *ACM Transactions on Database Systems*, 22(2):171–214, 1997.

[Col02] S. Colebourne. Joda home page. `http://joda.sourceforge.net`, current as of November, 2002.

[Cor02] IBM Corporation. International Components for Unicode (ICU). `http://oss.software.ibm.com/icu`, current as of May, 2002.

[CR87] J. Clifford and A. Rao. A Simple, General Structure for Temporal Domains. In *Proceedings of the Conference on Temporal Aspects in Information Systems*, pages 23–30, France, May 1987. AFCET.

[CSS94] R. Chandra, A. Segev, and M. Stonebraker. Implementing Calendars and Temporal Rules in Next Generation Databases. In *Proceedings of the International Conference on Data Engineering*, pages 264–276, February 1994.

[Dat88] C.J. Date. A Proposal for Adding Date and Time Support to SQL. *SIGMOD Record*, 17(2):53–76, June 1988.

[DELS00] C. E. Dyreson, W. S. Evans, H. Lin, and R. T. Snodgrass. Efficiently Supporting Temporal Granularities. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):568–587, 2000.

[DS98] C. E. Dyreson and R. T. Snodgrass. Supporting Valid-Time Indeterminacy. *ACM Transactions on Database Systems*, 23(1):1–57, 1998.

[Fal01] D. C. Fallside (editor). XML Schema Part 0: Primer. `http://www.w3c.org/TR/xmlschema-0`, current as of May, 2001.

[Fra87] J. Fraser. *Time the Familiar Stranger*. Tempus Books, Redmond, WA, 1987.

[Int00] International Organization for Standardization. Data elements and interchange formats – Information interchange – Representation of dates and times. Technical Report ISO8601:2000(E), ISO, December 2000.

[JC98] C. S. Jensen and C. E. Dyreson (editors). A Consensus Glossary of Temporal Database Concepts - February 1998 Version. In *Temporal Databases: Research and Practice, Lecture Notes in Computer Science 1399*, pages 367–405. Springer-Verlag, 1998.

[JS99] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, January/February 1999.

[KCGS96] L. Karttunen, J. Chanod, G. Grefenstette, and A. Schiller. Regular Expressions for Language Engineering. *Natural Language Engineering*, 2(4):305–238, 1996.

[KLS99] N. Kline, J. Li, and R. T. Snodgrass. Specifying Multiple Calendars, Calendric Systems, and Field Tables and Functions in TimeADT. Technical Report 41, TimeCenter, Aalborg, Denmark, May 1999.

[KSS96] S. Kraus, Y. Sagiv, and V. S. Subrahmanian. Representing and integrating multiple calendars. Technical Report CS-TR-3751, Univ. of Arizona, Dept. of Comp. Science, 1996.

[KT96] I. Kakoudakis and B. Theodoulidis. The TAU Temporal Object Model. Technical Report TR-96-4, TimeLab, University of Manchester (UMIST), 1996.

[Mic03] Sun Microsystems. Java 2 Platform, Standard Edition (J2SE) v. 1.4.1. `http://java.sun.com/j2se/1.4.1/`, current as of May, 2003.

[MMCR92] A. Montanari, E. Maim, E. Ciapessoni, and E. Ratto. Dealing with Time Granularity in the Event calculus. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, volume 2, pages 702–712, Tokyo, Japan, June 1992. ICOT.

[MS93] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[NB95] P. Navrat and M. Bielikova. Representing Calendrical Algorithms and Data in Prolog and Prolog III. *SIGPLAN Notices*, 30(7):45–51, 1995.

[NWJ02] P. Ning, X. S. Wang, and S. Jajodia. An Algebraic Representation of Calendars. *Annals of Mathematics and Artificial Intelligence*, 36(1-2):5–38, 2002.

[Ohl03a] H. J. Ohlbach. Project WebTNSS. http://www.pms.informatik.uni-muenchen.de/mitarbeiter/-ohlbach/WebTNSS/motivation.html, current as of May 2003.

[Ohl03b] H. J. Ohlbach. WebCal, an Advanced Calendar Server. `http://www.pms.informatik.uni-muenchen.de/mitarbeiter/ohlbach/WebTNSS/WebCal-WWW.pdf`, current as of May 2003.

[Pro03] Apache XML Project. Xalan-Java version 2.5.D1. `http://xml.apache.org/xalan-j`, current as of May, 2003.

[Sar93] N. Sarda. HSQL: A Historical Query Language. In *Temporal Databases: Theory, Design, and Implementation*, pages 110–140. Benjamin/Cummings, 1993.

[SJS95] M. D. Soo, C. S. Jensen, and R. T. Snodgrass. *An Algebra for TSQL2*, chapter 27, pages 505–546. Kluwer Academic Press, September 1995.

[SM99] C. M. Sperberg-McQueen. Regular Expressions for Dates. *Markup Languages*, 1(4):20–26, 1999.

[Sof02] CrystalClear Software. Boost Date-Time Library. `http://www.boost.org/libs`, current as of December, 2002.

[Soo93] M. D. Soo. Multiple Calendar Support for Conventional Database Management Systems. In R. T. Snodgrass, editor, *Proceedings of the Workshop on an Infrastructure for Temporal Databases*, pages FF1–FF17, June 1993.

[SS92] M. D. Soo and R. Snodgrass. Multiple Calendar Support for Conventional Database Management Systems. Technical Report 92-7, Computer Science Department, University of Arizona, February 1992.

[SSD+92] M. D. Soo, R. Snodgrass, C. Dyreson, C. S. Jensen, and N. Kline. Architectural Extensions to Support Multiple Calendars. TempIS Technical Report 32, Computer Science Department, University of Arizona, Revised May 1992.

[W3C00] W3C. Extensible Markup Language (XML) 1.0. http://www.w3c.org/TR/REC-xml, current as of October, 2000.

[WBBJ97] X. S. Wang, C. Bettini, A. Brodsky, and S. Jajodia. Logical Design for Temporal Databases with Multiple Granularities. *ACM Transactions on Database Systesms*, 22(2):115–170, 1997.

[WJL91] G. Wiederhold, S. Jajodia, and W. Litwin. Dealing with Granularity of Time in Temporal Databases. In *Proc. 3rd Nordic Conf. on Advanced Information Systems Engineering*, Trondheim, Norway, May 1991.

A An Example Calendar Specification

This appendix gives an example of a specification file for the Gregorian calendar. A calendar is mostly a collection of related granularities. Each granularity is specified by a <granularity> element. The element defines a granularity as a mapping from a (previously) defined granularity. The mapping can be regular (i.e., described by a formula), or irregular (implemented by a short piece of code). Code for the irregular mappings is supplied in the compiled Java class that supports the calendar. These specification files and others can be found at the τZAMAN project website (http://www.eecs.wsu.edu/~cdyreson/pub/tauZaman).

```
<!-- A calendar has one built-in granularity, the URL identifies calendar specific code -->
<calendarSpecification
  implUrl = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/GregorianCalendar.class"
  underlyingGranularity = "seconds">
  <!-- Minutes to seconds mapping, it is irregular because of leap seconds -->
  <granularity name = "seconds">
    <irregularMapping from = "minutes" relationship = "finer">
      <method name = "castMinuteToSecond" type = "cast"/>
    </irregularMapping>
  </granularity>
  <!-- Minutes to seconds mapping, it is irregular because of leap seconds -->
  <granularity name = "minutes">
    <irregularMapping from = "seconds" relationship = "coarser" >
      <method name = "castSecondToMinute" type = "cast" />
    </irregularMapping>
  </granularity>
  <!-- Hours to minutes is regular, each hour is 60 minutes -->
  <granularity name = "hours">
    <regularMapping from = "minutes" relationship = "coarser"
      periodSize = "60" groupSize = "60"/>
  </granularity>
  <!-- Mappings from days -->
  <granularity name = "days">
    <!-- Days to hours is regular, each day is 24 hours -->
    <regularMapping from = "hours" relationship = "coarser"
      periodSize = "24" groupSize = "24"/>
    <!-- Days to months is irregular -->
    <irregularMapping from = "months" relationship = "finer">
      <method name = "castMonthToDay" type = "cast" />
    </irregularMapping>
  </granularity>
  <!-- Weeks to days is regular, each week is 7 days -->
  <granularity name = "weeks">
    <regularMapping from = "days" relationship = "coarser"
      periodSize = "7" groupSize = "7"/>
  </granularity>
```

```
  <!-- Months to days is irregular, months have a different number of days -->
  <granularity name = "months">
    <irregularMapping from = "days" relationship = "coarser" >
      <method name = "castDayToMonth" type = "cast" />
    </irregularMapping>
  </granularity>

  <!-- Years to months is regular, each year has 12 months -->
  <granularity name = "years">
    <regularMapping from = "months" relationship = "coarser"
      periodSize = "12" groupSize = "12"/>
  </granularity>

</calendarSpecification>
```

Every specification file also has a `<descriptor>` element, which is not processed by the system and exists only for informative reasons. Here is the structure of a `<descriptor>`, listing its (optional) sub-elements, that exists in every specification file used for τZAMAN.

```
<descriptor>
  <versions>
    <currentVersion tag = "..." url = "..." />
    <previousVersion tag = "..." url = "..." />
  </versions>
  <contact>
    <name>
      <first>...</first>
      <middle>...</middle>
      <last>...</last>
    </name>
    <email>...</email>
  </contact>
  <reference>...</reference>
  <description>...</description>
</descriptor>
```

B An Example Calendric System Specification

This appendix gives an example calendric system specification file. A calendric system is a collection of calendars, plus inter-calendar granularity mappings. The mappings are implemented in a compiled Java class that is dynamically loaded when the calendric system object is created. The example specification is for a University of Arizona (UofA) calendric system. The system consists of the Gregorian calendar and a special calendar for the University. The UofA calendar is similar to the Gregorian calendar, but days are limited to times only when classes are in session, semesters and important dates within each semester (e.g., the dates of vacations like Spring Break).

```
<!-- A calendric system for use at the University of Arizona -->
<calendricSystem name = "UofALimitedCalendricSystem">

  <!-- The calendric system uses the Gregorian calendar -->
  <importCalendar name="ADGregorian"
    url="http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/ADGregorianCalendar.xml" />

  <!-- The calendric system also uses the UofA calendar -->
  <importCalendar name="UofACalendar"
    url="http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/UofACalendar.xml" />

  <!-- Try to parse dates in the specified calendar order -->
  <defaultInputOrder>
    ADGregorian
    UofACalendar
  </defaultInputOrder>

  <!-- Inter-calendar mappings -->
  <mappings>
```

```
    <!-- UofA hours to Gregorian hours is a congruent, irregular mapping since only some Gregorian hours are
UofA hours -->
    <irregularMapping
      from="hour" fromCalendar = "UofACalendar"
      to="hour" toCalendar = "ADGregorian"
      url="http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/UofALimitedCalendricSystem.class"
      relationship = "congruent" >
       <method name="castUofAHourToGregHour"/>
    </irregularMapping>
    <!-- Gregorian hours to UofA hours is also irregular -->
    <irregularMapping
      from="hour" fromCalendar = "ADGregorian"
      to="hour" toCalendar ="UofACalendar"
      url="http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/UofALimitedCalendricSystem.class"
      relationship = "congruent" >
       <method name="castGregHourToUofAHour"/>
    </irregularMapping>
  </mappings>
  <!-- Default regular expression to split up a temporal literal -->
  <defaults token="[a-zA-Z0-9]+" />

</calendricSystem>
```

C Available Properties

This appendix gives examples of each kind of property. We list and explain them individually. In the next section we provide an example of each. Additionally, we give all of the property operations available to a user to manage properties.

There are fourteen properties. Properties can be classified into three categories: format-related properties (twelve properties are of this kind), timezone-related properties (one property) and input priority properties (one property). The value of a format-related property is a sequence of templates used for input or output. The templates are applied in the order specified, until one matches.

Locale This property is in time-zone related class and it is used to specify a location for timezone displacement.

OverrideInputOrder This property specifies specifies which calendar to use to translate temporal literals, overriding the order in the calendar system specification.

InstantInputFormat Used to parse an instant temporal literal.

InstantOutputFormat Used to format an instant.

NowRelativeInstantInputFormat Used to parse a now-relative instant temporal literal.

NowRelativeInstantOutputFormat Formats a a now-relative instant during output.

IndeterminateInstantInputFormat For parsing an indeterminate instant.

IndeterminateInstantOutputFormat Format used in the output of an indeterminate instant.

PeriodInputFormat Used to parse a period.

PeriodOutputFormat Format used in the output of a period.

IntervalInputFormat Used to parse an interval.

IntervalOutputFormat Format used to output an interval.

IndeterminateIntervalInputFormat Used to parse an indeterminate interval temporal literal.

IndeterminateIntervalOutputFormat Format used to output an indeterminate interval.

D An Example Property Specification

This appendix gives an example property specification file, which contains all possible properties and therefore can be used as a full default property table by a TauZamanService with a related calendric system.

```
<!-- A property table is a collection of properties and field value tables -->
<propertyTable>

  <!-- Provides a mapping between the names (labels) and code for field value tables -->
  <fieldValueSupportMapper>
    <fieldvaluesupport label = "arabicNumeral"
     url = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/ArabicNumeral.class"/>
    <fieldvaluesupport label = "englishGregorianMonthNames"
     url = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/englishGregorianMonthNames.xml"/>
    <fieldvaluesupport label = "periodLeftDelimiterList"
     url ="http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/leftDelimiterList.xml"/>
    <fieldvaluesupport label = "periodRightDelimiterList"
     url ="http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/rightDelimiterList.xml"/>
    <fieldvaluesupport label = "directionList"
     url = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/directionList.xml"/>
    <fieldvaluesupport label = "englishNowNames"
     url = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/englishNowNames.xml" />
    <fieldvaluesupport label = "distributionNames"
     url = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/distributionNames.xml" />
  </fieldValueSupportMapper>

  <!-- The locale property is for establishing a timezone -->
  <property name = "Locale" value = "America/Los_Angeles" />

  <!-- This property sets up a format for parsing a determinate instant -->
  <property name = "InstantInputFormat" >
    <value>

      <format>
        <instant>
           <day value = "$day"/>
           <month value = "$month" />
           <year value = "$year" />
        </instant>
      </format>

      <fieldInfo variable = "month" name = "monthOfYear"
       using = "englishMonthNames" />
      <fieldInfo variable = "day" name = "dayOfMonth"
       using = "arabicNumeral" />
      <fieldInfo variable = "year" name = "year"
       using = "arabicNumeral" />

    </value>
  </property>

  <!-- The format for outputting a determinate instant -->
  <property name = "InstantOutputFormat" >
    <value>

      <format>
        <instant>
           $day, $month, $year
        </instant>
      </format>
```

```
    <fieldInfo variable = "month" name = "monthOfYear"
     using = "englishMonthNames" />
    <fieldInfo variable = "day" name = "dayOfMonth"
     using = "arabicNumeral" />
    <fieldInfo variable = "year" name = "year"
     using = "arabicNumeral" />

  </value>
</property>

<!-- The format for parsing an interval -->
<property name = "IntervalInputFormat">
  <value>

    <format>
      <months value = "$monthAsNumber" />
    </format>

    <fieldInfo variable = "monthAsNumber" name = "month"
     using = "arabicNumeral" />

  </value>
</property>

<!-- The format for outputting an interval -->
<property name = "IntervalOutputFormat">
  <value>

    <format>
      <months value = "$monthAsNumber" />
    </format>

    <fieldInfo variable = "monthAsNumber" name = "month"
     using = "arabicNumeral" />

  </value>
</property>

<!-- The format for parsing a now-relative instant -->
<property name = "NowRelativeInstantInputFormat">
  <value>

    <format>
       <now value = "$now" />
       <direction value = "$direction" />
       $interval
    </format>

    <fieldInfo variable = "now" name = "now"
     using = "englishNowNames"/>
    <fieldInfo variable = "direction" name = "directions"
     using = "directionList" />

    <importFormat variable = "interval" name = "IntervalInputFormat" />

  </value>
</property>

<!-- The format for outputting an now-relative instant -->
<property name = "NowRelativeInstantOutputFormat">
  <value>

    <format>
       <now value = "$now" />
       <direction value = "$direction" />
       $interval
    </format>

    <fieldInfo variable = "now" name = "now"
```

```
     using = "englishNowNames"/>
    <fieldInfo variable = "direction" name = "directions"
     using = "directionList" />

    <importFormat variable = "interval" name = "IntervalOutputFormat" />

  </value>
</property>

<!-- A period format involves two instants (indeterminate, now-relative, or determinate) -->
<property name = "PeriodInputFormat">
  <value>
    <format>
       <period>
       <delimiter value = "$leftClosed" />
       $instant
       $instant
       <delimiter value = "$rightClosed" />
       </period>
    </format>

    <fieldInfo variable = "leftClosed" name = "periodDelimiter"
     using = "leftDelimiterList"/>
    <fieldInfo variable = "rightClosed" name = "periodDelimiter"
     using = "rightDelimiterList"/>

    <importFormat variable = "instant" name = "InstantInputFormat" />

  </value>
</property>

<!-- The format for outputting a period, uses the instant format -->
<property name = "PeriodOutputFormat">
  <value>
    <format>
       <period>
       <delimiter value = "$leftClosed" />
       $instant
       $instant
       <delimiter value = "$rightClosed" />
       </period>
    </format>

    <fieldInfo variable = "leftClosed" name = "periodDelimiter"
     using = "leftDelimiterList"/>
    <fieldInfo variable = "rightClosed" name = "periodDelimiter"
     using = "rightDelimiterList"/>

    <importFormat variable = "instant" name = "InstantOutputFormat" />

  </value>
</property>

<!-- The format for parsing an indeterminate instant -->
<property name = "IndeterminateInstantInputFormat">
  <value>
    <format>
        <indeterminateInstant>
        $lower
        $upper
        <distribution value = "$distribution" />
        </indeterminateInstant>
    </format>

    <importFormat variable = "lower" name = "InstantInputFormat" />
    <importFormat variable = "upper" name = "InstantInputFormat" />

    <fieldInfo variable = "distribution" name = "distribution"
     using = "distributionNames"/>

  </value>
</property>
```

```
  <!-- The format for outputting an indeterminate instant -->
  <property name = "IndeterminateInstantOutputFormat">
    <value>
      <format>
          <indeterminateInstant>
          $lower
          $upper
          <distribution value = "$distribution" />
          </indeterminateInstant>
      </format>

      <importFormat variable = "lower" name = "InstantOutputFormat" />
      <importFormat variable = "upper" name = "InstantOutputFormat" />

      <fieldInfo variable = "distribution" name = "distribution"
       using = "distributionNames"/>

    </value>
  </property>

  <!-- Sets the calendar order in which literals are parsed (overrides calsys default) -->
  <property name = "OverrideInputOrder"
    value = "ADGregorian" />

</propertyTable>
```

E Example Field Value Specifications

This section has two examples of field value table specification files: `englishMonthNames.xml` and `leftDelimiterList.xml`.

The specification file for English month names relates Gregorian English month names with indexes. If indices are not explicitly given, by default they start from 1, and increment by one for each row in document order.

```
<fieldValueTable>

  <row string = "January" />
  <row string = "February" />
  <row string = "March" />
  <row string = "April" />
  <row string = "May" />
  <row string = "June" />
  <row string = "July" />
  <row string = "August" />
  <row string = "September" />
  <row string = "October" />
  <row string = "November" />
  <row string = "December" />

</fieldValueTable>
```

Indices can also be explicitly given with any row (with the exception that neither strings nor indexes can have duplicates in a single field value table). Below is an example of the `leftDelimiterList.xml` specification.

```
<fieldValueTable regex = "[^\s]">

  <row string = "[" value="1" />
  <row string = "(" value="2" />

</fieldValueTable>
```

The `regex` attribute of the `fieldValueTable` element specifies a *regular expression*, which is used to recognize a token of this type in a temporal literal. In this example, the regular expression specifies that any non-whitespace character could potentially be a left delimiter (alternatively, the regular expression could be shortened to just include '[' and '('). If no regular expression is given explicitly, as in `EnglishMonthNames`, a default regular expression, given in the calendric system specification, is used instead.

F Example Time Literals

This appendix has the time literals that are used to form instances of the temporal data types in the experiments described in Section 5. Although each of these literals is given in XML, we would like to remind the reader that τZAMAN can parse both XML and non-XML literals, such as the instant literal "March 6, 2003". Below are listed the example literals, formatted in XML.

```
<!-- Determinate instant -->
<instant>
  <day value = "6" />
  <month value = "March" />
  <year value = "2003" />
</instant>

<!-- Interval -->
<months value = "5" />

<!-- Now-relative instant -->
<now value = "now" />
<direction value = "+" />
<months value = "5" />

<!-- Indeterminate instant -->
<indeterminateInstant>
  <instant>
    <day value = "6" />
    <month value = "March" />
    <year value = "2003" />
  </instant>

  <instant>
    <day value = "6" />
    <month value = "March" />
    <year value = "2003" />
  </instant>

  <distribution value = "uniform" />
</indeterminateInstant>

<!-- Determinate period -->
<period>

  <delimiter value = "["/>

  <instant>
    <day value = "6" />
    <month value = "March" />
    <year value = "2003" />
  </instant>

  <instant>
    <day value = "6" />
    <month value = "March" />
    <year value = "2003" />
  </instant>

  <delimiter value = "]" />
</period>
```

```
<!-- Indeterminate period -->
<period>

  <delimiter value = "["/>

  <indeterminateInstant>
    <instant>
      <day value = "6" />
      <month value = "March" />
      <year value = "2003" />
    </instant>

    <instant>
      <day value = "6" />
      <month value = "March" />
      <year value = "2003" />
    </instant>

    <distribution value = "uniform" />
  </indeterminateInstant>

  <indeterminateInstant>
    <instant>
      <day value = "6" />
      <month value = "March" />
      <year value = "2003" />
    </instant>

    <instant>
      <day value = "6" />
      <month value = "March" />
      <year value = "2003" />
    </instant>

    <distribution value = "uniform" />
  </indeterminateInstant>

  <delimiter value = "]" />
</period>
```

In some of the experiments, subelements were used to wrap relevant information (rather than including the data as attribute values). We give only the literals for the determinate instant and now-relative instant.

```
<!-- Determinate instant -->
<instant>
  <day> 5 </day>
  <month> March </month>
  <year> 2003 </year>
</instant>

<!-- Now-relative instant -->
<now> now </now>
<direction> + </direction>
<months> 5 </months>
```

G Format Properties used in TAUZAMANTESTER

This appendix has the format properties used in Section 6. In Figure 22 an indeterminate instant is constructed (and output) according to an "IndeterminateInstantInput(Output)Format" as described in "property table 2" (the property table is chosen in the third drop-down menu in the center of the GUI). Since the "IndeterminateInstantInput(Output)Format" imports an "InstantInput(Output)Format", for completeness, we have to show both of the format properties. On the other hand, to keep it short, referenced field value tables will not be shown here.

```
<!-- The property table used in the TAUZAMANTESTER -->
<propertyTable>
```

```
<!-- Maps field value table labels to support code -->
<fieldValueSupportMapper>
  <fieldvaluesupport label = "arabicNumeral"
    url = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/ArabicNumeral.class"/>
  <fieldvaluesupport label = "englishMonthNames"
    url = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/englishMonthNames.xml"/>
  <fieldvaluesupport label = "distributionNames"
    url = "http://www.eecs.wsu.edu/~cdyreson/pub/tauzaman/release/distributionNames.xml" />
</fieldValueSupportMapper>

<!-- Input property for instants -->
<property name = "InstantInputFormat" >
  <value>
    <format><date month = "$month" year = "$year" day = "$day" /></format>
    <fieldInfo variable = "month" name = "monthOfYear" using = "englishMonthNames" />
    <fieldInfo variable = "day" name = "dayOfMonth" using = "arabicNumeral" />
    <fieldInfo variable = "year" name = "year" using = "arabicNumeral" />
  </value>
</property>
<!-- Output property for instants -->
<property name = "InstantOutputFormat" >
  <value>
    <format><date> month = "$month" year = "$year" day = "$day" </date></format>
    <fieldInfo variable = "month" name = "monthOfYear" using = "englishMonthNames" />
    <fieldInfo variable = "day" name = "dayOfMonth" using = "arabicNumeral" />
    <fieldInfo variable = "year" name = "year" using = "arabicNumeral" />
  </value>
</property>
<!-- Input property for indeterminate instants -->
<property name = "IndeterminateInstantInputFormat">
  <value>
    <format>
      <lower> $lower </lower>
      <upper> $upper </upper>
      <distribution value = "$distribution" />
    </format>
    <!-- Each bound is a determinate instant. -->
    <importFormat variable = "lower" name = "InstantInputFormat" />
    <importFormat variable = "upper" name = "InstantInputFormat" />
    <!-- The default distribution is uniform.
    <fieldInfo variable = "distribution" name = "distribution"
     using = "distributionNames" />
  </value>
</property>

<!-- Output property for indeterminate instants -->
<property name = "IndeterminateInstantOutputFormat">
  <value>
    <format>
      <support> <lower> $lower </lower> </support>
      <support> <upper> $upper </upper> </support>
      <distribution value = "$distribution" />
    </format>
    <importFormat variable = "lower" name = "InstantOutputFormat" />
    <importFormat variable = "upper" name = "InstantOutputFormat" />
    <fieldInfo variable = "distribution" name = "distribution"
     using = "distributionNames" />
  </value>
</property>
```

In Figure 23 a determinate period is constructed (and output) according to an “PeriodInput(Output)Format” in property table 1. Since “PeriodInput(Output)Format” imports an “InstantInput(Output)Format”, for completeness, we have to show both of the format properties. Property table 1 used in the TAUZAMANTESTER is the same as the example property table shown in Appendix D. So, “PeriodInput(Output)Format” and “InstantInput(Output)Format” that it imports can be found there.