TAUZAMAN: A SYSTEM FOR SUPPORTING MULTIPLE CALENDARS
ON THE WEB

by

BEDIRHAN URGUN

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

May 2003
To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of BEDIRHAN URGUN find it satisfactory and recommend that it be accepted.

________________________________________
Chair
ATTRIBUTION

This thesis is the draft of a journal paper that has been co-authored by the following people. Bedirhan Urgun is the lead author. The other authors (in order of the journal submission) and their affiliations are listed below.

- Curtis E. Dyreson, School of EECS, Washington State University, Pullman, WA, 99164, cdyreson@eecs.wsu.edu.
- Nick Kline Microsoft Corporation, Redmond, WA, nkline@microsoft.com
- Jessica K. Miller Department of Computer Science, University of Arizona, Tucson, AZ 85719, jessica@cs.arizona.edu
- Richard T. Snodgrass Department of Computer Science, University of Arizona, Tucson, AZ 85719, rts@cs.arizona.edu
- Michael D. Soo Amazon.com, Seattle, WA, soo@amazon.com
- Christian S. Jensen Department of Computer Science, Aalborg University, Fr. Bajers Vej 7E, Dk-9220 Aalborg Øst, Denmark, csj@cs.auc.dk

The authors made several passes over the material, each refining and adding to the prose so it is difficult to identify sole contributors. The lead author, Bedirhan Urgun, however, wrote the first draft of much of the paper, and was the lead designer for the extended components of rZaman. There is a website that supports the paper, and lists all the contributors to the project: www.eecs.wsu.edu/~cdyreson/pub/tauZaman.
TAUZAMAN: A SYSTEM FOR SUPPORTING MULTIPLE CALENDARS ON THE WEB

Abstract

by Bedirhan Urgun, M.S.
Washington State University
May 2003

Chair: Curtis Dyreson

The goal of the $\tau$Zaman project is to build a native Java system for formatting and manipulating times and dates in multiple calendars and languages. The project name is composed of the Turkish word for time, Zaman (pronounced Zah-mon), and the Greek letter, $\tau$ (pronounced tau), which denotes that it is part of the Temporal Access for Users (tau) project, started at the University of Arizona. $\tau$Zaman builds on two earlier projects, called Multical and timeADT. $\tau$Zaman adds dynamic loading of calendars, a client/server architecture, XML specification of calendars and calendric systems, XML formatting for dates and times, and a robust, portable implementation in the Java programming language. This thesis describes the architecture of $\tau$Zaman and experimentally quantifies the cost of using a calendar server to translate and manipulate dates.
TABLE OF CONTENTS

LIST OF TABLES ......................................................... vii

LIST OF FIGURES ....................................................... viii

CHAPTER 1  Introduction ................................................ 1

1.  Motivation ......................................................... 4

2.  \(\tau\text{Zaman Concepts} \) ........................................ 9

2.1.  Calendars ....................................................... 9

2.2.  Calendric Systems ............................................. 13

2.3.  Temporal Data Types ......................................... 14

3.  \(\tau\text{Zaman Architecture} \) .................................... 19

3.1.  Overview ....................................................... 19

3.2.  Supporting Operations on Temporal Data Types ............. 24

3.3.  Calendar Support .............................................. 27

3.4.  Input/Output ................................................... 34

3.5.  The TauZamanSystem and Client/Server Package ........... 37

4.  Coding Statistics and Experimental Results .................... 42

4.1.  Coding Statistics ............................................. 43

4.2.  Experiment Environment ..................................... 43

4.3.  Experiments on TauZamanService Initialization ............ 44

5.  Graphical User Interface (GUI) Tool ............................ 49

6.  Related Work ..................................................... 52
7. Conclusions and Future Work ........................................... 57

REFERENCES ................................................................. 60
1. An Example Calendar Specification ................................. 63
2. An Example Calendric System Specification .................. 66
3. An Example Property Specification .............................. 67
4. Available Properties ..................................................... 75
5. Example Field Value Specifications ............................. 78
6. Example Time Literals ................................................... 80
7. Format Properties used in GUI Tool ............................ 84
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Common Calendars</td>
<td>10</td>
</tr>
<tr>
<td>2.</td>
<td>Calendar Properties</td>
<td>13</td>
</tr>
<tr>
<td>3.</td>
<td>Examples of temporal data types</td>
<td>14</td>
</tr>
<tr>
<td>4.</td>
<td>Valid arithmetic expressions and results</td>
<td>17</td>
</tr>
<tr>
<td>5.</td>
<td>Proposed Set of Comparison Operators</td>
<td>18</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Converting a date from a USA to a European format</td>
<td>6</td>
</tr>
<tr>
<td>2.</td>
<td>An XML-based conversion from USA to European date format</td>
<td>6</td>
</tr>
<tr>
<td>3.</td>
<td>A time value is translated from English to Hindi</td>
<td>7</td>
</tr>
<tr>
<td>4.</td>
<td>A Gregorian calendar to Islamic calendar conversion</td>
<td>7</td>
</tr>
<tr>
<td>5.</td>
<td>Converting between local and remote τZaman servers</td>
<td>8</td>
</tr>
<tr>
<td>6.</td>
<td>Converting days to months</td>
<td>8</td>
</tr>
<tr>
<td>7.</td>
<td>Converting months to days</td>
<td>9</td>
</tr>
<tr>
<td>8.</td>
<td>Evaluating an &quot;earlier than&quot; predicate</td>
<td>9</td>
</tr>
<tr>
<td>9.</td>
<td>Adding an interval to an instant</td>
<td>10</td>
</tr>
<tr>
<td>10.</td>
<td>The Russian Calendric System</td>
<td>14</td>
</tr>
<tr>
<td>11.</td>
<td>Overview of the τZaman system architecture</td>
<td>21</td>
</tr>
<tr>
<td>12.</td>
<td>Main classes of temporal data type, timestamp support and their interactions in τZaman</td>
<td>25</td>
</tr>
<tr>
<td>13.</td>
<td>Main classes of calendar support and their interactions in τZaman.</td>
<td>28</td>
</tr>
<tr>
<td>14.</td>
<td>Calendar Repository structure</td>
<td>30</td>
</tr>
<tr>
<td>15.</td>
<td>The property stack structure, P is an instance of a property</td>
<td>32</td>
</tr>
<tr>
<td>16.</td>
<td>Detailed internal and rough external class interactions of Input/Output package</td>
<td>34</td>
</tr>
<tr>
<td>17.</td>
<td>An example instant input format property</td>
<td>35</td>
</tr>
<tr>
<td>18.</td>
<td>An example indeterminate instant input format property</td>
<td>37</td>
</tr>
<tr>
<td>19.</td>
<td>An example indeterminate instant literal</td>
<td>37</td>
</tr>
<tr>
<td>20.</td>
<td>Field list structure for the indeterminate instant</td>
<td>38</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>21. Detailed internal and rough external class interactions of TauZamanSystem and Client/Server package.</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>22. A TauZamanSystem after initialization</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>23. Communicating with multiple services and calendric systems</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>24. The client/server architecture of τZaman</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>25. A diagram of the environment for the experiments</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>26. Average loading times (in milliseconds) of a TauZamanLocalService</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>27. Average loading time through a remote service</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>28. Average input/output times (in milliseconds) for different kinds of temporal literals using a local service</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>29. Average input/output times (in milliseconds) for different kinds of temporal literals using a remote service</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>30. A snapshot of τZaman GUI tool in which a indeterminate instant is processed.</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>31. A snapshot of τZaman GUI tool in which a determinate period is processed.</td>
<td>51</td>
<td></td>
</tr>
</tbody>
</table>
τZaman: a System for Supporting Multiple Calendars on the Web

Bedirhan Urgun*  Curtis E. Dyreson†  Nick Kline‡  Jessica K. Miller§
Richard T. Snodgrass¶  Michael D. Soo‖  Christian S. Jensen***

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, for instance an Islamic calendar, yet can interoperate with dates and times in other calendars, say a Gregorian calendar. τZaman is a native Java system for formatting and manipulating times and dates in multiple calendars and languages. 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. The most important design feature of τZaman is that it separates the calendar-independent and calendar-dependent components while retaining the ability to format times in multiple calendars and languages. Calendars are modularly specified in XML-formatted files and can be dynamically loaded into a running τZaman system. τZaman also supports the parsing and output of times in XML, so users have a great deal of flexibility for specifying and representing times. 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 relativity theory posits that an observer measures time relative to a frame of reference. For most observers, especially those traveling at a 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, there are about major forty calendars in daily use [Fra87]. Even though time is measured, represented, and used in many different ways, many applications impose a single interpretation for time and temporal operations. For instance, most modern database management systems require dates to be represented in the Gregorian calendar.

*School of EECS, Washington State University, Pullman, WA, 99164, burgun@eeecs.wsu.edu
†School of EECS, Washington State University, Pullman, WA, 99164, cdyreson@eeecs.wsu.edu
‡Microsoft Corporation, Redmond, WA, nkline@microsoft.com
§Department of Computer Science, University of Arizona, Tucson, AZ, 85719, jessica@cs.arizona.edu
¶Department of Computer Science, University of Arizona, Tucson, AZ, 85719, rts@cs.arizona.edu
‖Amazon.com, Seattle, WA, soo@amazon.com
***Department of Computer Science, Aalborg University, Fr. Bajers Vej 7E, Dk–9220 Aalborg Øst, Denmark, csj@cs.au.dk
This paper presents \( \tau \)Zaman, a system that has the ability to support multiple calendars and provide calendar-independent temporal operations. The project name is composed of the Turkish word for time, \textit{Zaman}, (pronounced zah-mon), and the Greek letter, \( \tau \) (pronounced \textit{tau}), which denotes that it is part of the Temporal Access for Users (tau) project started at the University of Arizona. \( \tau \)Zaman is an enhancement of two earlier systems: Multical [MS92] and timeADT [KLS]. 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.

\( \tau \)Zaman extends Multical and timeADT in several directions.

- \( \tau \)Zaman is a client/server system, unlike Multical and timeADT. 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 would be useful to share the calendar among many applications and users. A client/server system enables the creation of “calendar servers” that can talk to multiple clients. We anticipate that there will be \( \tau \)Zaman servers running on well-known sites, especially for the major calendars.

- \( \tau \)Zaman makes extensive use of the Extensible Markup Language (XML) [W3C00]. The increasing popularity of formatting data in XML for applications on the web makes it desirable to port the system to a web-based, XML platform. In Multical and timeADT, the calendar-related specification files are unstructured text files. In \( \tau \)Zaman they are in XML. Using XML helps to improve the parsing and validation of the specification files, making it easier to develop calendars. Additionally, we anticipate that times and dates formatted in XML will become common in the future. Multical and timeADT have no special support for parsing times and dates in XML. \( \tau \)Zaman adds this important feature.

- It should be possible to easily incorporate new calendars and other user-defined information, such as languages or input-output formats for temporal constants, into a multi-calendar system. But in Multical and timeADT, the entire system has to be recompiled to add or change a single calendar or language. A key part of the design of \( \tau \)Zaman is the ability to add calendars and input-output formats on the fly, at run-time.

- Multical and timeADT are implemented in C and C++, respectively. In \( \tau \)Zaman we chose to use Java, for five reasons. First, portability is a big concern. We’d like \( \tau \)Zaman to operate on most machines, even PDAs. The Java Virtual Machine (JVM) provides a stable, platform-independent environment in which \( \tau \)Zaman can be run. Second, Java is “network-friendly” in the sense that it has strong support for network communication and building client/server systems. Java’s Remote Method Invocation (RMI) classes provide efficient and reliable support for building distributed systems architecture. Ideally, we would like \( \tau \)Zaman to run as a calendar server to provide a network resource for handling times in a specific calendar, such as the Gregorian or Islamic calendars. 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 \( \tau \)Zaman, such as temporal constants and calendar definition files, will be formatted in XML. \( \tau \)Zaman benefits from the widely-used and reliable XML parsing and processing packages of Sun’s “Java 2 platform Standard Edition (J2SE)” [SM03]. Fifth, Java also supports dynamic class loading. Dynamic class loading can be used to extend a calendar server with new calendars at run-time.
The rest of the paper is organized as follows. The next section presents several example scenarios showing how $\tau$Zaman can be used. Section 3 presents the major time-related concepts that are implemented in $\tau$Zaman. The architecture is described in Section 4. The description consists of an overview of the major packages, and a detailed discussion of the roles of individual classes. We performed several experiments to measure the efficiency of $\tau$Zaman. The results are presented in Section 5. Section 6 presents a preliminary tool, with Graphical User Interface (GUI), that uses $\tau$Zaman to translate and manipulate dates both by utilizing local $\tau$Zaman and a $\tau$Zaman server. The last two sections discuss related research and list the contributions of this research.

## 2 Motivation

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

The first scenario is a user, let’s call her Leslie, who 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 short Perl script, or a program in another string processing language to perform the conversion. But $\tau$Zaman can also convert times between formats. To do a format conversion, Leslie would first connect to a Gregorian calendar $\tau$Zaman 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 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.

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. $\tau$Zaman 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. 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. $\tau$Zaman supports using different languages for *fields* in formats, such as the name of the month. New tables for language-support 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. A single $\tau$Zaman system can load several calendars at once and apply inter-calendar conversions. $\tau$Zaman 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 $\tau$Zaman that has a reliable implementation of the Islamic calendar. Leslie runs a “remote” $\tau$Zaman server for the Gregorian calendar. The “client” API for $\tau$Zaman does not differ substantially between local and remote servers, so clients do not have to be specialized to manage local and remote services differently. The figure shows a client in contact with a single remote server, but in general, a $\tau$Zaman client can simultaneously
communicate with multiple $\tau$Zaman servers.

The fifth scenario examines a time granularity conversion [WBBJ97, DELS00]. 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 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. An indeterminate time is a time that is not known precisely. Figure 7 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. $\tau$Zaman supports both intra- and inter-calendar granularity conversions. Additionally, $\tau$Zaman provides classes that model indeterminate times, so the indeterminacy can be accounted for (or discarded if desired) in the conversion.

The sixth 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 Australia 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 8, and an example arithmetic operation is depicted in Figure 9. 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. $\tau$Zaman provides a complete set of temporal comparison operations and a useful set of arithmetic operations. $\tau$Zaman 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 some 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 \( \tau \)-Zaman Concepts

This section introduces concepts that are of utility to users of \( \tau \)-Zaman, 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 group of calendars. \( \tau \)-Zaman supports temporal operations on three temporal data types: instants, periods, and intervals. 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 \( \tau \)-Zaman architecture to support the concepts.

3.1 Calendars

A calendar is a human abstraction of time. 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.

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 example calendars.

We emphasize that the usage of a calendar depends on the cultural, legal, and even business orientation of the user. For example, business enterprises generally perform accounting relative to some fiscal year. However, the definition of fiscal year varies depending on the enterprise. 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.

The intrinsic characteristics of a calendar define the intrinsic semantics of the calendar or 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, \( \text{isLeapYear(year)} \), 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. For instance, in the Gregorian calendar, birth dates are typically measured in or known to the granularity of days and train schedules to that of minutes. Since measurements are discrete, a granularity creates a discrete image of a continuous time-line. More precisely, the underlying, continuous 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 modular collection of related granularities. 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. 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* may be considered to be 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”. The same date may also be expressed in different 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 [ISO0]. 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.

3.2 Calendric Systems

Calendric systems are defined as collections of calendars where each calendar covers a contiguous and non-overlapping portion of the time-line, called an epoch. 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 10 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., 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.

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 A.D. to 9999 A.D. [Dat88, MS93]. This is inadequate for some applications, such as developing a historical record of ancient Egypt, that fall outside of this epoch. Also, applications that use time values that are within this epoch, but in a different calendar cannot be adequately supported. By allowing multiple calendars to exist within an application, and supporting calendric systems with multiple calendars, we provide a general notion of time that is able to express the entire history of an enterprise.
Zaman 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 modelled 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locale</td>
<td>Location for timezone displacement</td>
</tr>
<tr>
<td>Instant input format</td>
<td>Input format string for instants; there are also formats for now-relative and indeterminate instants.</td>
</tr>
<tr>
<td>Instant output format</td>
<td>Output format string for instants; there are also formats for now-relative and indeterminate instants.</td>
</tr>
<tr>
<td>Interval input format</td>
<td>Input format string for interval; there is also a format for indeterminate intervals</td>
</tr>
<tr>
<td>Interval output format</td>
<td>Output format string for interval; there is also a format for indeterminate intervals</td>
</tr>
<tr>
<td>Period input format</td>
<td>Input format string for periods</td>
</tr>
<tr>
<td>Period output format</td>
<td>Output format string for periods</td>
</tr>
</tbody>
</table>

Table 2: Calendar Properties
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 modelled 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 contiguous subset of the time domain. Usually, a period is 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 $g_1 \leq g_2$. For example, in the Gregorian calendar the 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. 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, when observed at finer granularities do not necessarily have a fixed duration. For example, the length of the interval “one month” in 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, negatively infinite and positively infinite 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 [CDI1997]. 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*.

\(\tau\text{Zaman}\) also has support for 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.

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operator</th>
<th>Operand 2</th>
<th>Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>interval</td>
<td>interval</td>
<td></td>
</tr>
<tr>
<td>interval</td>
<td>+</td>
<td>interval</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>−</td>
<td>interval</td>
<td>interval</td>
</tr>
<tr>
<td>instant</td>
<td>+</td>
<td>interval</td>
<td>instant</td>
</tr>
<tr>
<td>instant</td>
<td>−</td>
<td>interval</td>
<td>instant</td>
</tr>
<tr>
<td>interval</td>
<td>+</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td>instant</td>
<td>−</td>
<td>instant</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>∗</td>
<td>numeric</td>
<td>interval</td>
</tr>
<tr>
<td>numeric</td>
<td>∗</td>
<td>interval</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>/</td>
<td>numeric</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>/</td>
<td>interval</td>
<td>numeric</td>
</tr>
<tr>
<td>interval</td>
<td>+</td>
<td>period</td>
<td>period</td>
</tr>
<tr>
<td>period</td>
<td>+</td>
<td>interval</td>
<td>period</td>
</tr>
<tr>
<td>period</td>
<td>−</td>
<td>interval</td>
<td>period</td>
</tr>
</tbody>
</table>

Table 4: Valid arithmetic expressions and results.

It is important to note that the operations are not orthogonal. For example, *instant* ∗ *instant* is undefined since no reasonable semantics for the expression exists.

\(\tau\text{Zaman}\) 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 set of period relations and showed that the set was complete in the sense that all possible relationships between two periods were covered [All83]. \(\tau\text{Zaman}\) extends Allen’s operators with an analogous set of operators for the instant and interval data types. Table 5 lists the available operations in \(\tau\text{Zaman}\). 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, \(\tau\text{Zaman}\) 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 calendar *days* is to be added to an instant, say “12:00, March, 1 2003” at Gregorian calendar *hours*. Below are four reasonable semantics for evaluating the operation.

**Mismatch** Give a mismatched granularity error [ABQPdO85].

**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.
Table 5: Proposed Set of Comparison Operators

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operator</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>=</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>&lt;</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>&gt;</td>
<td>interval</td>
</tr>
<tr>
<td>instant</td>
<td>=</td>
<td>instant</td>
</tr>
<tr>
<td>instant</td>
<td>precedes</td>
<td>instant</td>
</tr>
<tr>
<td>instant</td>
<td>precedes</td>
<td>period</td>
</tr>
<tr>
<td>instant</td>
<td>overlaps</td>
<td>period</td>
</tr>
<tr>
<td>period</td>
<td>precedes</td>
<td>instant</td>
</tr>
<tr>
<td>period</td>
<td>overlaps</td>
<td>instant</td>
</tr>
<tr>
<td>period</td>
<td>precedes</td>
<td>period</td>
</tr>
<tr>
<td>period</td>
<td>=</td>
<td>period</td>
</tr>
<tr>
<td>period</td>
<td>meets</td>
<td>period</td>
</tr>
<tr>
<td>period</td>
<td>overlaps</td>
<td>period</td>
</tr>
<tr>
<td>period</td>
<td>contains</td>
<td>period</td>
</tr>
</tbody>
</table>

**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.

4 **τZaman Architecture**

The key design features of our 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.

In the remainder of this section we present the architecture for τZaman. We first give a broad overview of the major packages and how they are related to each other and to users.

4.1 **Overview**

Figure 11 shows the major components of the architecture. In the figure, each box represents a group of related package(s), whereas, users are represented with ovals. A directed edge, whether with a solid or dashed line, indicates that the source, a package or a user, makes use of the target package’s methods. There are two main flows of operations that can be done by using τZaman. One is related to configuration of τZaman, such as loading of calendars and properties or setting up τZaman services. As one of the main aspects of τZaman, these and many other configurations can be performed dynamically. So, configuration
flow does not happen once, at the beginning of the execution, but possibly many times during the execution. The other flow of operation is related to temporal operations and input/output. Once temporal data types, such as instant, interval or period, are constructed (input), τZaman provides a set of operations that users can perform on them. Nevertheless, these two flows are represented in the same figure, Figure 11, differentiated by using dashed lines for system configuration flow and solid lines for temporal operations and input/output.

The figure identifies the user level and low level calendar-independent aspects (Temporal Data Types and Timestamps respectively), the calendar-related aspects (Calendar, Calendric System, Property and Field), the merger of them (Input and Output), and the user level system configuration interface (TauZamanSystem, Client/Server).

The Temporal Data Type package 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 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 11. 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. The construction uses a calendar. 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 only interact with TauZamanSystem and Client/Server packages for input and output as shown in Figure 11. For temporal operations, Temporal Data Type package interacts with Timestamp package. A constructed temporal data type contains one or more granules, and timestamp provides temporal operations on granules.

The calendric system, calendar, property, and field packages manage access to calendar-related services. A person who designs new calendars, or modifies existing calendars will interact with these packages via TauZamanSystem and Client/Server package. 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 11 shows that the Input/Output packages using both Timestamp and calendar-related packages. I/O bridges the user dependent and independent parts of \(\tau\)Zaman. This is because 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. The granules form 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. Here calendars and related structures, such as properties, are located in the calendric system, calendar, property, and field packages, on the other hand, granules are located in Timestamp package.

The TauZamanSystem packages provides calendar dependent support to end users. As stated this support has two direction. First, TauZamanSystem is used to perform input/output operations by Temporal Data Type package. Second, as shown in Figure 11 a user can configure calendar dependent structures by communicating directly to TauZamanSystem. Loading new calendric systems and properties are some examples of this configuration support. At this point, one may ask the question of “why are temporal data types not constructed or output directly by using Input/Output packages?”. In other words, “why is there another layer, that is TauZamanSystem and Client/Server package, between Temporal Data Type and Input/Output packages?”. \(\tau\)Zaman brings the notion of services. A user may have different services with which he/she can create temporal data types. Since different services can include different calendric systems, it is important to know which temporal data type belongs to which service, that is which temporal data type is created by using a particular service. To prevent this ambiguity this additional layer of TauZamanSystem is used between Temporal Data Type and Input/Output packages. Next paragraph explain the service notion in \(\tau\)Zaman in more detail.

\(\tau\)Zaman can be run as either a server, a client, or both, as described in more detail below.

**Server**  A TauZamanSystem can be set as a (remote) server. The server provides calendar resources to clients on a network. Clients communicate with the server using remote procedure call (RPC). \(\tau\)Zaman 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. The server manages all the calendar-related information, while the client maintains instances of the temporal data types. This means that temporal arithmetic and comparison operations can be performed at the client-side; the server is involved only on input or output, and granularity conversions. A server can support multiple clients. Each client has a separate property space, managed by the server.

**Client**  A (remote) client can connect to a rich collection of calendar services provided by a remote \(\tau\)Zaman server. A client can connect to multiple servers; clients individually manage each server connection as a separate object. All client/server communication is done using RPC.

**Local**  \(\tau\)Zaman 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 a local service can still connect as a client to other remote servers.

The rest of this section provides a discussion of individual components introduced here. We first present the low-level building blocks in \(\tau\)Zaman, such as components for supporting operations on temporal data types and calendar-related services. Next, the high-level components are described, in particular, The TauZamanSystem and Client/Server package. Each component is represented with individual figures. In the figures, each solid box represents a class and dashed box represents a package. A directed edge indicates that
4.2 Supporting Operations on Temporal Data Types

The classes for temporal data types and timestamps form the calendar-independent part of \( \tau \text{Zaman} \). These classes and interactions between them are shown in Figure 12. From the same figure, it is also clear that temporal data types and timestamp classes use services provided by \( \tau \text{ZamanSystem} \) and \( \text{Client/Server Package} \) as shown in Figure 11.

The core of this part is the TimeValue class. TimeValue encapsulates the 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). 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, \textit{beginning} and \textit{forever}, representing the earliest and latest possible times, respectively. We used Java’s \texttt{long} data type for a time, so \( 2^{64} \) different times can be represented. In sixty-four bits it is possible to represent the lifetime of the universe, sixteen billion years, to the granularity of nanoseconds (\( 10^{-9} \) seconds).

Each granularity creates a discrete image of the time-line as a sequence of granules. The Granule class associates a TimeValue with a granularity to form a granule. For instance, the TimeValue representing 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 modelling 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 probability mass function
describes the probability of each indeterminate alternative. Common mass functions, such as uniform,
poisson, etc., are provided. Finally, a now-relative granule 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. These classes implement the semantics discussed in Section 3.3, for instance, an Interval can
be added to an Instant, but two Instants cannot be added. String constructors are also provided (which call
Input/Output). Instant, Period and Interval can represent determinate, indeterminate, and now-relative times.
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 ~ March, 29 2003”, and
an Instant with a now-relative granule from “now + 5 days”.

Along with normal arithmetic operators, Allen’s work on period relationships between entities [All83]
forms the basis of the temporal comparison operators that we support. But with granularity present there are
several possible semantics for performing each 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.
To support various semantics, operations using temporal data types are provided in a Semantics interface.
The interface is implemented with whatever semantics is desired by the user. The Semantics is further
subclass into DeterminateSemantics and IndeterminateSemantics. The only difference between the two
kinds of Semantics is that the DeterminateSemantics returns boolean values for comparison operations,
but the IndeterminateSemantics returns an ExtendedBoolean value. ExtendedBoolean implements a three-
valued logic on the values true, false, and maybe.

4.3 Calendar Support

This section presents calendar-dependent classes in ℰ\(\text{Zaman}\). These classes are used to load, activate
and deactivate calendric systems, calendars, properties and field values, to convert temporal constants to
timestamps, and to perform granularity conversions. Figure 13 shows individual components that composes
calendar support and interactions between them. Classes in calendar support do not use any other major
components of ℰ\(\text{Zaman}\) as shown in Figure 11. Unlike other individual figures that explain major components
in Figure 11 in detail, Figure 13 has a different structure. In the figure, since calendar-dependent classes
does not make use of other major packages, there are no dashed boxes representing packages. On the
other hand, since calendar dependent classes are composed of several sub-packages, such as calendar, calendric
system, property and field, dashed ovals are used to populate classes that are in the same sub-packages.
Solid directed edges represent intra-package interactions, whereas, dashed directed edges represent inter-
package interactions between classes.

4.3.1 Calendar

A calendar is a collection of granularities and mappings between them. 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 ℰ\(\text{Zaman}\) is that is 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.
Figure 13: Main classes of calendar support and their interactions in τZaman.
The second information set is the location of Java classes that provide the code to do irregular granularity mappings. An irregular mapping is a special kind of conversion that is not reducible to a simple formula. Most granularity conversions are regular, and can be described completely in the XML specification. 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 \( \tau \)Zaman. Irregular mappings though need special code. 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. A calendar developer has to provide a Java class, which is dynamically loaded, to perform an irregular mapping.

With the two information sets, \( \tau \)Zaman can load everything it needs about a new calendar, provided the calendar specification file is valid. An exception is thrown if the specification is invalid or other problems are detected during loading.

\( \tau \)Zaman uses a calendar repository to share calendars among multiple users. To prevent duplicate loading of calendars and increase the performance of \( \tau \)Zaman, 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, \( \tau \)Zaman provides a calendar “refresh” operation to force reloading of a calendar when desired, for instance, if the specification file has been updated. Figure 14 shows the structure of individual calendars with their parent calendar repository. 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.
4.3.2 Calendric System

Like calendars, calendric systems are also described by XML specification files. The calendric system specification files provide definitions for epochs, calendars, a description of how to integrate multiple calendars, default properties (see Section 4.3.3), 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 Java class that performs the mapping. The class is loaded during loading of the calendric system.

Calendric systems are shared in a repository. To prevent duplicate loading of calendric systems, $\tau$Zaman 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.

4.3.3 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 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.

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 C. 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.

Properties, unlike calendars and calendric systems, are different for each individual user. Users want specific formats for input and output of temporal constants. To illustrate this, assume that a user first wants instants to be parsed according to a “$mm/dd/yyyy$” format. Later the user decides to change the format to “$dd/mm/yyyy$”. The user could also change other formatting features, like the language. To support user-specific properties, for each user $\tau$Zaman allocates a private property stack for each property. Figure 15 shows the stacks of properties within a user’s private property space. Since the properties have calendar-related components, the stacks are maintained on the server-side, rather than by a $\tau$Zaman client. When a new property is desired, the user asks a $\tau$Zaman 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 deactivate the property, causing the property to be popped from the stack.

Within $\tau$Zaman the PropertyManger class handles the management of properties via PropertyStackService class. We identified six different operations on properties, which simplify the management from the user point of view. Appendix C lists the valid operations that can be applied to properties.
4.3.4 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 understand whether the period is closed or open.

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 that map strings to field values. Consider the literal “Mar/20/2003”. After parsing, the month field would have the value “Mar”. A language table would be used to map the string to the value 3 representing the month of March. During output the language tables are used to replace field values with the appropriate output string.

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

4.4 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. Since output is largely the reverse of input, we will present only the process for input in detail. Figure 16 shows the classes and their interactions in Input/Output package. In addition, it also shows inter-package interactions in parallel with Figure 11.

Input is somewhat complicated because there are multiple calendars, languages, and a variety of format properties, but basically it boils down to five major steps. The rest of this section explains these steps in detail.

1. Parse the format, build a DOM.
2. Parse the literal, build a DOM.
3. Match the literal DOM with the format DOM.

4. Extract field values from the literal 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 Document Object Model (DOM) for the format it contains. A format specifies an acceptable skeleton or structure for a temporal literal. Figure 17 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"/>
```

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 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 following literal is DOM-equivalent to the one given above.

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

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

Figure 18: An example indeterminate instant input format property

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

Figure 19: An example indeterminate instant literal

appear where some text is expected. If no format matches, then the literal cannot be parsed by the property. An exception is thrown indicating the 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 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 whitespace between elements 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 language table that has all of the possible legal field values. For example the $month field uses the EnglishMonthName language 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 be a non-zero sequence of alphabetic characters from the Western character set. 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.

The fifth step puts the legal, string values matched by each field variable into a field list structure. The field list is a collection of information extracted from a temporal literal. A calendar will convert the field list into a granule. When a field value is placed into the field list, one final transformation is performed. The string value is converted to a number. The transformation is a simple lookup in the language table; the table maps legal field values to the corresponding field numerals.

Indeterminate instants, now-relative instants, and determinate and indeterminate periods all have “bounding” instants. The instant input/output format property can be imported into the format properties for other temporal data types. Figure 18 shows an example indeterminate instant format property. The upper and lower support are both instants. So the instant input format is imported to parse those components in an indeterminate instant literal, such as the literal shown in Figure 19. 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 20.
4.5 The TauZamanSystem and Client/Server Package

The TauZamanSystem class is the manager for access to $\tau$Zaman. Figure 21 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 drawn.

A user, whether it is a client or a server, creates a single TauZamanSystem. When $\tau$Zaman is run as a server, the TauZamanSystem is responsible for communicating with clients, and managing the many repositories. Figure 22 shows the structure of a TauZamanSystem and underlying repositories after system is created. The repositories are populated over time as a client loads calendric systems, calendars, properties, and language tables. The choice of setting a system as a client or a server is application-dependent. Client/server communication in $\tau$Zaman is designed to use the Remote Procedure Call (RPC) technology in Sun’s Java Remote Method Invocation (RMI) package.

The TauZamanSystem also provides TauZamanServices, which is the 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 deactivate properties, and to input and output temporal literals. To increase the flexibility of the system for the users, an end-user may have several TauZamanServices, connecting the client to a local system or multiple remote servers. The TauZamanSystem stores the currently active service; clients switch among the many services by setting the active service to the desired service.

Each TauZamanService has an active state that stores the current set of active components of a TauZa-
manService. 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 Astronomy and Gregorian calendars, and in the same service also load the Russian calendric system, which manages the Gregorian and Communist calendars. Only one calendric system however 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 23 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 TauZamanService and active CalendricSystem to the constructor, the active service and state are retrieved within the constructor using class methods in the TauZamanSystem class. This keeps the constructor parameter lists short. Furthermore, one of our design goals was to reimplement Multical and timeADT operations. Neither of these systems had client/server services, nor could they communicate with multiple calendric systems. The temporal data type operations in \( \tau \text{Zaman} \), even with the additional functionality, are the same as their Multical and timeADT counterparts. In \( \tau \text{Zaman} \), 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 TauZamanService and CalendricSystem.

There were two main design criteria that guided the development of the server/client functionality in
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 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 and field list structure classes are serializable.

The first design criteria lead TauZamanService to be specialized into two services, TauZamanLocalService and TauZamanRemoteService. TauZamanRemoteServices are designed to be remote objects, registered with the object registry, and can be referenced by a client system. To set up a TauZamanSystem as a server, the user should first register and public the URL of a separate remote object, which is called TauZamanRemoteServiceHandler. Any TauZamanSystem knowing this URL can connect to the server as a client.

Figure 24 illustrates the client/server structure. In the figure a TauZamanSystem at the client side connects to the server’s TauZamanRemoteServiceHandler through a TauZamanRemoteService 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 TauZamanSystem, on both the client and server side, can also have other TauZamanServices.

5 Coding Statistics and Experimental Results

This section reports on the size and performance of τZaman. The performance of local and remote τZaman services are measured, for initial loading of a system and input and output of temporal literals. The rest of this section is organized as follows. The statistics about the τZaman implementation are given first, followed by the performance experiments. We describe the experiment environment, the results of an experiment fo-
cusing on the TauZamanService initialization process, and the results of experiments related to input/output. Each experiment also compares the performance of local and remote τZaman services.

5.1 Coding Statistics

τZaman is a moderately large system. It consists of approximately 12,500 lines of Java code, not including the code in system or third-party supplied classes. We compiled 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 depends on several packages. The packages are listed below along with the tasks for which we used them.

- **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.

5.2 Experiment Environment

We conducted the experiments in a distributed system environment because τZaman is a client/server system. Figure 25 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**.

5.3 Experiments on TauZamanService Initialization.

A client accesses τZaman by creating a TauZamanService, 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 C. 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). Figure 26 gives the measured times. We averaged the times over five tests to smooth the effects of network congestion. The times varied by as much as 40 milliseconds per test.

The initial loading time is, not surprisingly, much longer than subsequent loading times because τZaman provides repositories to cache reused objects. But even the initial loading time is a “one-time” cost.

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**
Figure 25: A diagram of the environment for the experiments

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Calendric System</th>
<th>Property Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial loading</td>
<td>633</td>
<td>518</td>
<td>115</td>
</tr>
<tr>
<td>Consecutive loadings</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 26: Average loading times (in milliseconds) of a TauZamanLocalService
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 Figure 27. 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 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.

However, there is about a 30 millisecond difference between the measured loading times at client side and at server side. Factoring in the 8 milliseconds of network, round-trip time the additional overhead is because a TauZamanRemoteService object is marshalled and unmarshalled 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 TauZamanServices.

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 C 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. Figure 25 reports the results of the experiment. Like the other experiments it is separated into two different measurements: an initial loading (for the first input/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 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.

The final observation about the result of this experiment is that the ranking of the kinds of temporal literals by initial I/O times is different from the ranking by consecutive I/O. The reason is somewhat subtle. During an input operation, \( \tau \)Zaman tries to construct, in order, a determinate instant, a now-relative instant, and finally an indeterminate instant. Each kind of instant has a different input format property. So an indeterminate instant is constructed only after parsing fails for a determinate and now-relative instant. Initially,
the ranking differs because the cost of fetching the field value tables dominates the cost of parsing.

We next tested input/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 marshalling of parameters, so the overall cost should be greater than that of the local service. Figure 29 shows the results of this experiment.

The cost of the initial loading include 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 on the remote communication. The times in Figure 28 are lower than those in Figure 29. The last observation to make about the results is that now-relative and indeterminate instants are even more expensive. The reason is that each parse failure results in another round of RPC between the client and the server until the appropriate kind of instant is finally constructed.

6 Graphical User Interface (GUI) Tool

Along with the Application Programming Interface (API) that τZaman v.1 provides for programmers, we also produced a preliminary test tool with GUI that performs Input/Output operations. Some of the reasons for such a visual tool can be listed as,

- to evaluate the idea of producing a tool, which uses τZaman v.1 as its underlying source.
to provide a user friendly interface for Input/Output operations, which deals with formats and xml fragments.

- to simplify the configuration details of $\tau$Zaman for a user by the help of user friendly components. In other words, with the tool, for example, users can activate a TauZamanService by selecting it from a combobox instead of writing a code piece.

As one of our future works, we intend to improve this tool to include all of the capabilities of the $\tau$Zaman.

When run, tool initializes its own TauZamanSystem. It then gets two TauZamanServices; a TauZamanLocalService and a TauZamanRemoteService. All the critical information, urls of calendric systems, property tables and remote $\tau$Zaman server come from a specialized xml file and therefore can be changed easily. Calendric system used is same as given calendric system specification file placed in Appendix B. One of the calendars, Gregorian calendar, that it imports can be found in Appendix A. Format properties that the tool uses are placed in Appendix G.

After initializing services, a panel, Input/Output panel, appears to get user instructions to perform various Input/Output operations. Input/Output panel, which is shown in Figure 30, includes three sub-panels.

- Input sub-panel includes a scrollable text area, in which user can input his/her xml time values.

- Configuration sub-panel provides a set of GUI components by which user can configure his/her TauZamanSystem to perform Input/Output operations. For example, user can activate the remote or local $\tau$Zaman service by selecting the appropriate one from a combobox. Similarly user can choose desired property table to use for input/output formatting and calendric system for time value processing. User can also select desired temporal data type of the time value entered in the text area of input sub-panel. Configuration sub-panel also includes a label that shows the processing time for Input/Output operation, as we had in Coding Statistics and Experiments section.

- Output sub-panel includes a scrollable text area, in which user can see xml output of his/her xml time values entered in the text are of input sub-panel.

When user types a time value in input sub-panel, configures the settings and presses the “PROCESS” button in configuration sub-panel, the output will appear in the output panel. On the other hand, should any exception occurs during processing, a general error dialog box shows up.

Two snapshots of the tool are shown in Figure 30 and Figure 31. In Figure 30, an indeterminate instant is entered in input sub-panel. To process this time value and output it according to another format, TauZamanSystem is configured by using the components in configuration sub-panel. Figure 30 also shows the output. In Figure 31, a determinate period time value is processed and outputted to the output sub-panel.

As stated at the beginning of this section, this tool does not provide full functionality of $\tau$Zaman v.1. On the other hand, it captures the future functionalities of $\tau$Zaman, which are bound to provide a user friendly environment for end users. Some of the important future functionalities are,

- dynamic configuration of services, such as adding a new service, removing an existing service or setting its TauZamanSystem as a server.

- management of properties such that property stack of each calendric system is visible and easy to update.

- management of temporal data types in a way which allows to choose them to perform a temporal operation.
Figure 30: A snapshot of $\tau$Zaman GUI tool in which a indeterminate instant is processed.

Figure 31: A snapshot of $\tau$Zaman GUI tool in which a determinate period is processed.
7 Related Work

Related research can be broadly classified into two categories: modeling and implementation. 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.

Anderson described a formal framework to support conceptual time spaces using inheritance hierarchies [And82, And83]. Her model also supports multiple conceptual times. Zaman can be considered as a practical extension 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 inconsistent with domains of larger units. For example, weeks are inconsistent with months 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.

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.

Bielikova and Navrat argue for declarative, rather than an algorithmic, calendar definitions [BN96]. Algorithmic definitions sometimes lead to oversimplification of predictions for future times and unnecessary approximations of past times [BN96]. For example, in the Islamic calendar, determining the first day of the month of Ramadan can not be predicted by an algorithm, although an approximation exists and be used by some cultures. Bielikova and Navrat address 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.

Ning, Wang and Jajodia propose an algebraic approach to define granularities and calendars [NWJ02]. They argue 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, Zaman uses specialized code. We are investigating using their approach to support irregular mappings in Zaman.

Kraus, Sagiv and Subrahmanian propose 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 [KSS97]. They also show how to support multiple calendars and argue that specifying a time point as integers or real numbers is cumbersome for human beings. However, at the user level we achieved to represent temporal constants as symbolic representation using temporal constant-timestamp conversions.

A subset of our project is presented by Kakoudakis and Theodoulidis by restricting the temporal model to support only Gregorian calendar and use limited number of granularities in Gregorian calendar [KT96].

Another effort to support multiple calendars is done by Chandra, Segev and Stonebraker. An informative comparison between theoretical background of their work and Multical, which has the theoretical background of our project can be found in Chandra et. al [CSS94].

In the second branch, the application arena, several systems with support for temporal data types, temporal operations, datetime calculations and conversions exist. Most of the applications on the web that perform datetime calculations and conversions are limited in scope, have static calendar support, usually with four or five different calendars, and very limited datetime format representation. On the other hand, there are also applications that support multiple calendar handling and temporal data type operations.
Boost [CS02] is a date-time library in C++, which supports three basic temporal data types, point, duration and interval, and operations on them with three valued boolean logic as our project does. It has iterators on time and date ranges, which helps user to go through days of a week, for example. On the other hand, although Boost is claimed to handle multiple calendars, it supports them in a static context, in other words calendars are not loaded dynamically to the system, and do not have generic inter-calendar conversions and calculations. Also one of its design goals is to support ISO 8601 compliant input/output representation, which although is an international standard for time representation, limits user expressive flexibility.

The International Components for Unicode (ICU) [C+02] is a set of libraries, which are developed by 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 it addressed 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 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 calendar in static context and ISO 8601 compatible inputs and outputs. Additionally it allows user to form their formats for date time representation through an API, which has the advantage to check the correctness of produced format before it is tried on inputs. In our project formats can be produced in xml files and thus can be shared and examined easily. In Joda to relate a temporal data type, let’s say an instant, to a calendar, user has to pass calendar object to instant constructor, which we achieved to get rid of for the sake of clarity. Moreover, it supports period and interval temporal data types under the name of TimePeriod, which we believe makes distinction transparent from user point of view. And lastly Joda does not include temporal conversions between different temporal data types but additionally argues the immutability of temporal data types for being safe in thread environments.

WebCal [Ohl03], a calendar server produced by OhlBach, is a client/server architecture for providing temporal support. WebCal is a part of the WebTNSS [Ohl] 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-relativity.
- For performance reasons, τZaman can be used as a local service without a need of a server, but WebCal solely a server.
- WebCal does have language support, such as “May, 2, 2003”, partly because it is designed to provide calendar independent time representation for WebTNSS project. On the other hand, τZaman supports language dependent formats in time values.

Finally, there are several papers strictly devoted to the parsing of dates.
Karttunen et al. [KCGS96] proposes a regular expression calculus for natural language applications. And as one of its illustrations, they form 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 their own formats 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 Karttunen et al.’s proposal is very broad.

Sperberg-McQueen [SM99] argues that recognition of dates is possible by regular expressions and gives a lex code, which recognizes and validates ISO 8601 complaint dates.

Cameron [Cam99] provides a set of XML shallow parsing regular expressions, which can be used to parse an XML document into individual items, such as attribute and text values. He argues that this style of parsing is relatively faster than off-the-shelf XML parsing and processing tools. If we have pursued a complete character based parsing of XML inputs, then this technique would have been a valuable source to use. On the other hand, since in our project useful information is stored in either text nodes or attribute values and since generally date inputs are short, we pursued a DOM model parsing to get the text and attribute values. One another problem with using regular expressions for the whole XML fragments 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.

8 Conclusions and Future Work

τZaman is a native Java system 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. It builds upon the earlier Multical and timeADT projects, but extends those projects in significantly new directions. 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.

- Repositories allow effective sharing of components. τZaman uses repositories to enable sharing of critical data, such as calendars, calendric systems, granularity mappings, and 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.

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

- τZaman has 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.

- Wider access to calendar-related services with a client/server model that makes calendars available on a network.

In future we plan to extend this research in several directions. One direction is studying how to craft user interfaces to ease the task and reduce the possibility of mistakes by calendar developers. A tool to visualize
granularities and graphically construct granularity mappings would greatly benefit developers, as would a point-and-click tool for creating input and output formats. A second direction is to refine and extend the mappings between granularities to include granularities with holes. In this context, detailed experiments on temporal operations and conversions between different granularities will be performed. One client that we are interested in supporting is the 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. Finally, we’d like to replace Java’s current calendar support with \( \tau \)Zaman. This will help to demonstrate the extensibility and representation of \( \tau \)Zaman.

**References**


A An Example Calendar Specification

This appendix gives an example of a specification file for the Gregorian calendar.

```
<calendarSpecification underlyingGranularity = "second"
     implUrl = "http://www.eecs.wsu.edu/~burgun/GregorianCalendar.class" >

   <granularity name = "second">
      <irregularMapping from = "minute" relationship = "finer">
         <method name = "castMinuteToSecond" type = "cast" />
      </irregularMapping>
   </granularity>

   <granularity name = "minute">
      <irregularMapping from = "second" relationship = "coarser">
         <method name = "castSecondToMinute" type = "cast" />
      </irregularMapping>
   </granularity>

   <granularity name = "hour">
      <regularMapping from = "minute" relationship = "coarser">
         <method name = "castHourToMinute" type = "cast">
            <periodSize = "60" groupSize = "60" />
         </method>
      </regularMapping>
   </granularity>

   <granularity name = "day">
```
<regularMapping from = "hour" relationship = "coarser"
    periodSize = "24" groupSize = "24"/>

<irregularMapping from = "month" relationship = "finer">
    <method name = "castMonthToDay" type = "cast" />
</irregularMapping>
</granularity>

<granularity name = "week">
    <regularMapping from = "day" relationship = "coarser"
        periodSize = "7" groupSize = "7"/>
</granularity>

<granularity name = "month">
    <irregularMapping from = "day" relationship = "coarser">
        <method name = "castDayToMonth" type = "cast" />
    </irregularMapping>
</granularity>

<granularity name = "year">
    <regularMapping from = "month" relationship = "coarser"
        periodSize = "12" groupSize = "12"/>
</granularity>
</calendarSpecification>

Along with critical elements in every specification file, such as granularity elements in calendar speci-
ification, there is also a descriptor element, which is not processed by the system and exists for informative
reasons. Here is the structure of a descriptor element, with all its sub-elements, that exists in every specifi-
cation file used for $\tau$Zaman.

<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.
C An Example Property Specification

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

<propertyTable>

<!-- optional helper labels -->
<fieldValueSupportMapper>
  <fieldvaluesupport label = "arabicNumeral"
    url = "http://www.eecs.wsu.edu/~burgun/ArabicNumeral.class"/>
  <fieldvaluesupport label = "englishMonthNames"
    url = "http://www.eecs.wsu.edu/~burgun/englishMonthNames.xml"/>
  <fieldvaluesupport label = "periodLeftDelimiterList"
    url = "http://www.eecs.wsu.edu/~burgun/periodLeftDelimiterList.xml"/>
  <fieldvaluesupport label = "periodRightDelimiterList"
    url = "http://www.eecs.wsu.edu/~burgun/periodRightDelimiterList.xml"/>
  <fieldvaluesupport label = "directionList"
    url = "http://www.eecs.wsu.edu/~burgun/directionList.xml"/>
</fieldValueSupportMapper>

</propertyTable>
<fieldvaluesupport label="englishNowNames"
    url="http://www.eecs.wsu.edu/~burgun/englishNowNames.xml" />
<fieldvaluesupport label="distributionNames"
    url="http://www.eecs.wsu.edu/~burgun/distributionNames.xml" />
</fieldValueSupportMapper>

<property name="Locale" value="Pullman" />

<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>

<property name="IntervalInputFormat">
    <value>
        <format>
            <months value="$monthAsNumber"/>
        </format>
    </value>
</property>
</format>

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

<property name = "IntervalOutputFormat">
  <value>
    <format>
      <months value = "$monthAsNumber" />
    </format>
    <fieldInfo variable = "monthAsNumber" name = "month"
      using = "arabicNumeral" />
  </value>
</property>

<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>

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

</value>
</property>

<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>

<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>

<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>

<property name = "IndeterminateInstantOutputFormat">
  <value>
    <format>
      <indeterminateInstant>
        $lower
        $upper
        <$distribution value = "$distribution" />
        </indeterminateInstant>
      </format>
    </value>
  </property>

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

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

</value>
</property>

<property name = "OverrideInputOrder"
  value = "http://www.eecs.wsu.edu/~burgun/ADGregorianCalendar.xml" />

</propertyTable>

D Available Properties

This appendix gives examples of each kind of property are given. Here we list and explain them individually. Additionally we give all property operations available to user to coordinate properties. As mentioned previously there are 14 kinds of properties. And properties can be classified into three; format-related properties (12 properties are of this kind), timezone-related properties (1 property) and input priority properties (1 property).

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

OverrideInputOrder  This property is in input priority class and value in this property uniquely specifies a calendar (by using its specification file URL) in the current calendric system that will be used to translate temporal constants.

42
InstantInputFormat Set of information, which specify mechanisms to parse a given instant temporal literal in XML.

InstantOutputFormat Set of information, which specify mechanisms to format a given instant timestamp to temporal literal in XML.

NowRelativeInstantInputFormat Set of information, which specify mechanisms to parse a given now relative instant temporal literal in XML.

NowRelativeInstantOutputFormat Set of information, which specify mechanisms to format a given now relative instant timestamp to temporal literal in XML.

IndeterminateInstantInputFormat Set of information, which specify mechanisms to parse a given indeterminate instant temporal literal in XML.

IndeterminateInstantOutputFormat Set of information, which specify mechanisms to format a given indeterminate instant timestamp to temporal literal in XML.

PeriodInputFormat Set of information, which specify mechanisms to parse a given period temporal literal in XML.

PeriodOutputFormat Set of information, which specify mechanisms to format a given period timestamp to temporal literal in XML.

IntervalInputFormat Set of information, which specify mechanisms to parse a given interval temporal literal in XML.

IntervalOutputFormat Set of information, which specify mechanisms to format a given interval timestamp to temporal literal in XML.

IndeterminateIntervalInputFormat Set of information, which specify mechanisms to parse a given indeterminate interval temporal literal in XML.

IndeterminateIntervalOutputFormat Set of information, which specify mechanisms to format a given indeterminate interval timestamp to temporal literal in XML.

Here are the list of property operations that can be executed by the user at run-time. There are six operations that a user can execute on properties.

**Property Activation** Pushes a given property into its private stack. For example, if being activated property is of type “InstantInputFormat”, then it will be pushed into stack, which uniquely exists for “InstantInputFormat” properties. An “active property” defined as “the property, which is at the top of its property stack”.

**Property Deactivation** Pops a property given its name from its private stack. For example, if the name given for this operation is “InstantInputFormat”, then a property will be removed from the top of the stack, which uniquely represents “InstantInputFormat”. This operation can also be defined as “deactivation of active property of a given property name”. There is only one exception when deactivating the active property and that is, the active property, which is the last property in its stack can not be popped or deactivated. In other words, in a property stack, there should always be at least one property, and the first properties, which are pushed to and be never popped (until the end of program) from their stacks are called default properties.
**Property Deactivate All** By using definition of “Property Deactivation”, this operation deactivates all active properties in every individual property stack.

**Property Set Default** Pops all properties until the last one from the stack, which represents a given property name. In other words, by using definition for “deactivate”, executing “deactive” for a given property stack.

**Property Set Default All** Pops all properties until the last ones form all the stacks. In other words, by using definition for “deactivate all”, executing “deactive all” for all properties in every individual property stack.

**Property Get Image:** Prints (property name, specification file URL) pairs for all properties, in a given property stack. For example, given “InstantInputFormat”, it gives all the keys of properties that are in the “InstantInputFormat” stack.

### E Example Field Value Specifications

Here we give two examples of field value support specification files, namely; `englishMonthNames.xml` and `leftDelimiterList.xml`.

Specification file for `englishMonthNames`, 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.

```xml
<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 number (with the exception that neither strings nor indexes can have duplicates in a single field value table).

The specification file for a `directionList` consists of the direction of time, possibly in a now-relative instant.

```xml
<fieldValueTable regex = "[^\s]">
  <row string = "[" />
  <row string = "]" />
</fieldValueTable>
```
Indices here are again implicit. On the other hand, regex attribute of fieldValueTable element contains a regular expression, which will be used to fetch related delimiter information when a temporal constant is being parsed. If not exists, as in EnglishMonthNames, a default regular expression given in calendric system specification file will be used, instead.

F Example Time Literals

In this appendix we placed all the time literals that are used in experiments to form instances of the temporal data types.

First string literals, which use attributes over elements. Some of the string literals may not be perceived as XML, however, we allow this to have inputs such as “March, 23, 2003” as well. So, this inputs and their corresponding formats in properties will be wrapped by an element. This also means that, although, inputs may not conform “single root” condition of XML structure, they have to conform other rules, such as “attribute values can not contain XML”, etc.

Determinate Instant:

<instant>
  <day value = "5" />
  <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 = "5" />
    <month value = "March" />
    <year value = "2003" />
  </instant>

  <instant>
    <day value = "5" />
    <month value = "March" />
    <year value = "2003" />
  </instant>

  <distribution value = "uniform" />
</indeterminateInstant>
Determinate Period:

<period>

<delimiter value = "["/>

<instance>
    <day value = "5" />
    <month value = "March" />
    <year value = "2003" />
</instance>

<instance>
    <day value = "5" />
    <month value = "March" />
    <year value = "2003" />
</instance>

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

Indeterminate Period:

<period>

<delimiter value = "["/>

<indeterminateInstant>
    <instant>
        <day value = "5" />
        <month value = "March" />
        <year value = "2003" />
    </instant>

    <instant>
        <day value = "5" />
        <month value = "March" />
        <year value = "2003" />
    </instant>

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

<indeterminateInstant>
    <instant>
        <day value = "5" />
        <month value = "March" />
        <year value = "2003" />
    </instant>

    <instant>
        <day value = "5" />
    </instant>

</period>
Second we give time literals, which use elements over attributes, for only Instant temporal data type and its Now Relative Instant derivative.

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 GUI Tool

In this appendix we give the format properties used in section 6. In Figure 30 an indeterminate instant is constructed (and output) according to an “IndeterminateInstantInput(Output)Format” in property table 2. Since “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.
<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 = "arabicNumerals" />
    <fieldInfo variable = "year" name = "year" using = "arabicNumerals" />
  </value>
</property>

<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 = "arabicNumerals" />
    <fieldInfo variable = "year" name = "year" using = "arabicNumerals" />
  </value>
</property>

<property name = "IndeterminateInstantInputFormat">
  <value>
    <format>
      <lower> $lower </lower>
      <upper> $upper </upper>
      <distribution value = "$distribution" />
    </format>
    <importFormat variable = "lower" name = "InstantInputFormat" />
    <importFormat variable = "upper" name = "InstantInputFormat" />
    <fieldInfo variable = "distribution" name = "distribution" using = "distributionNames" />
  </value>
</property>

<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 31 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 gui tool is same as the example property table placed in Appendix C. So, “PeriodInput(Output)Format” and “InstantInput(Output)Format” that it imports can be found there.