JSC-61777-models/environment/time

JSC Engineering Orbital Dynamics Time Representations Model

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

Package Release JEOD v5.1

Document Revision 1.2.2
July 2023

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

JSC-61777-models/environment/time

JSC Engineering Orbital Dynamics Time Representations Model

Document Revision 1.2.2
July 2023

Gary Turner

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

Abstract

The Time Representations Model in JEOD v5.1 represents a complete reformulation from the UTIME structure seen in JEOD1.x. Like its predecessor, it is responsible for tracking time as a simulation advances; unlike its predecessor, it can do so in any number of time representations, using any number of clocks, that may be ticking at different rates from one another.

The Time Representations Model accurately represents the most commonly used reference times throughout a simulation, and includes the capacity to control additional user-specified time references.

Contents

Chapter 1

Introduction

1.1 Purpose and Objectives of the Time Representations Model

The Time Representations Model in JEOD v5.1 represents a complete reformulation from the UTIME structure seen in JEOD1.x. Like its predecessor, it is responsible for tracking time as a simulation advances; unlike its predecessor, it can do so in any number of time representations, using any number of clocks, that may be ticking at different rates from one another.

While the SI second is a well-defined quantity, counted to high precision by atomic clocks, and incorporated into many physical constants, it is not always the best clock to use for modeling purposes. For example, when modeling the rotation of Earth, it is preferable to use a sidereal clock (1 day = 1 rotation), rather than the synodic clock (1 day = 1 solar passage) associated with the SI second. The length of the SI second was chosen such that there would be approximately 86,400 seconds in one synodic day, and the passage of the SI second is represented by International Atomic Time (TAI). However, the length of the synodic day, now associated with Universal Time (UT1), rarely lasts 86,400 SI seconds, and fluctuates with little predictability. Earth-based civil-time, or Universal Coordinated Time (UTC), attempts to closely match the passage of UT1, while ticking at the same rate as TAI.

Further complicating the management of time are other clocks with various epochs (points in time where their value was 0). Mission-Elapsed-Time (MET) is often used to track time since a mission began, and the Global Positioning System (GPS) has its own clock. Users may wish to run only in local time, rather than UTC, or have clocks that can output their data in two or more local times for international missions.

In JEOD 1.x, there was a tacit assumption that time always advanced forward. In this implementation, we can run time in reverse to find, for example, a set of initial conditions that produce a desired end-state. As we look toward making JEOD applicable to interplanetary missions, it may be desirable to incorporate a solar-based timescale, with capability for relativistic corrections; that capability has also been implemented.

1.2 Context within JEOD

The following document is parent to this document:

- *JSC Engineering Orbital Dynamics* [2]

The Time Representations Model forms a component of the environment suite of models within JEOD v5.1. It is located at models/environment/time.

1.3 Documentation History

Author	Date	Revision	Description
Gary Turner	November 2009	1.0	JEOD 2.0.0 release
Gary Turner, Blair Thompson	March 2010	1.1	JEOD 2.0.1 release Addition of clarifying text New instructions for obtaining default data
Gary Turner	April 2011	1.2	Addition of comments regarding TimeStandard and edits for change from Time to JeodBaseTime
Gary Turner	January 2012	1.2.1	Addition of comments regarding Julian Date inclusion for release 2.2.
Gary Turner	January 2014	1.2.2	Addition of comments regarding initialization and setting epochs for UDEs
Gary Turner	January 2014	1.3	Addition of comments regarding lead seconds

1.4 Documentation Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [3] and is organized into the following chapters:

Chapter 1: Introduction – This introduction contains four sections: purpose and objective, context within JEOD, document history, and document organization.

Chapter 2: Product Requirements – Describes requirements for the Time Representations Model.

Chapter 3: Product Specification – Describes the underlying theory, architecture, and design of the Time Representations Model in detail. It will be organized in four sections: Conceptual Design, Mathematical Formulations, Detailed Design, and Version Inventory.

Chapter 4: User Guide – Describes how to use the Time Representations Model in a Trick simulation. It is broken into three sections to represent the JEOD defined user types; Analysts or users of simulations (Analysis), Integrators or developers of simulations (Integration), and Model Extenders (Extension).

Chapter 5: Verification and Validation – Contains Time Representations Model verification and validation procedures and results.

Chapter 2

Product Requirements

Requirement time_1: Top-level requirement

Requirement:
This model shall meet the JEOD project requirements specified in the JEOD v5.1 top-level document.

Rationale:
This model shall, at a minimum, meet all external and internal requirements applied to the JEOD v5.1 release.

Verification:
Inspection time_1 on page 79.

Requirement time_2: Physical Time

Requirement:
The Time Representations Model shall include a representation of time that can be used for integration of the dynamic vehicle state. It shall be capable of propagating forward, and in reverse.

Rationale:
Not all time representations accurately represent the standard definition of "second", which is required for the integration of the physical state. There should be one clearly identified representation that is always used for integration purposes. To identify a suitable state that is capable of creating a particular outcome, it is necessary to be able to integrate from the outcome back in time to the original state.

Verification:
Test time_1 on page 79, test time_2 on page 80, and test time_11 on page 93.

Requirement time_3: Necessary Clocks

Requirement:

The Time Representations Model shall be able to represent time in each of the following time systems:

3.1 TAI
International Atomic Time, a very accurate and stable time scale calculated as a weighted average of the time kept by about 200 cesium atomic clocks in over 50 national laboratories worldwide.

3.2 UT1
Universal Time, a measure of the rotation angle of the Earth as observed astronomically.

3.3 UTC
Coordinated Universal Time, the basis for the worldwide system of civil time.

3.4 MET
Any number of Mission Elapsed Times, each with distinct user-defined epochs.

3.5 GPS
The time system used by the Global Positioning System.

3.6 TT
Terrestrial Time, the time system used by the Ephemeris models.

3.7 GMST
Greenwich Mean Sidereal Time, the time system used to simulate the rotation of Earth.

3.8 TDB
Barycentric Dynamic Time, potentially useful for higher fidelity ephemeris models, or for inner-solar-system mission.

3.9 UDE
Any number of clocks each set to tick with any of the other clocks, but each with a distinct user-defined epoch.

Rationale:

The purpose of the time module is to represent time in the multiplicity of time scales that are expected to be needed by various simulation developers.

Verification:

Test time_3 on page 83.

Requirement time_4: Necessary Representation Formats

Requirement:

The Time Representations Model shall be able to represent time in the each of the following formats:

4.1 Julian Date

4.2 Modified Julian

4.3 Truncated Julian Time

4.4 Seconds since epoch

4.5 Calendar / Clock
Provide a calendar where a calendar is defined, including year, month, day, hour, minute, second.

Provide just a clock (comprising day, hour, minute, second) where a calendar is not defined.

Rationale:
Different models require different formats by which time is input to their respective methods. This is the compilation of the formats determined to be the most useful for external models. A request to add centuries since epoch was deferred due to the ambiguous definition of 'century' (being 100 years, or 36,525 days). Instead, days since epoch was added, this can be utilized to obtain centuries since epoch using whatever definition of 'century' the model requires.

Verification:
Test time_6 on page 87, test time_7 on page 88, and test time_8 on page 88.

Requirement time_5: Time Initialization by Representation

Requirement:
The Time Representations Model shall be capable of initializing time using any of the following time representations:

- GPS
- MET
- TAI
- TDB
- TT
- UDE
- UT1
- UTC

Rationale:
The purpose of the time module is to represent time in the multiplicity of time scales that are expected to be needed by various simulation developers; this implies the internal ability to convert easily from one to another.

Verification:
Test time_4 on page 85, test time_5 on page 86 and time_8 on page 88.

Requirement time_6: Time Initialization by Format

Requirement:

The Time Representations Model shall be capable of initializing time using any one of the following formats:

6.1 Calendar
Using a calendar-based format yyyy/mm/dd::hh:mm:ss when such a format is well defined in the appropriate time representation

6.2 Clock
Using a clock-based format dd::hh:mm:ss when a larger calendar format is not well defined.

6.3 Truncated Julian Time
Days since the NASA Truncated Julian Time epoch.

6.4 Modified Julian Time
Days since the Modified Julian Time epoch.

6.5 Julian Time
Days since the Julian Time epoch.

6.6 Seconds since epoch
Seconds since the appropriate epoch (e.g. J2000, or a User-defined epoch).

6.7 Days since epoch
Days since the appropriate epoch (e.g. J2000, or a User-defined epoch).

Rationale:

While conversion between these types is algorithmic, it can be time consuming and potentially prone to error. The provision of these capabilities allows the user to focus on getting the simulation dynamics correctly configured without worrying about whether some known time has been correctly converted into a usable input.

Verification:

Test time_6 on page 87, test time_7 on page 88, and test time_8 on page 88.

Requirement time_7: Time Initialization Bypass

Requirement:

The Time Representations Model must be capable of bypassing the initialization of time-types.

Rationale:

There are applications where no fixed reference time is necessary, and arbitrarily inputting some time is, at best, redundant.

Verification:

Test time_1 on page 79

Requirement time_8: Initialization of User-Defined-Epoch times

Requirement:

The Time Representations Model must be capable of initializing User-Defined-Epoch times (such as Mission Elapsed Time) in the following ways:

8.1 UDE value and fixed-epoch value known
The simulation is initialized relative to some fixed epoch, and the value of the UDE time is known at the start of the simulation. The UDE epoch can be determined.

8.2 UDE epoch and fixed-epoch value known
The simulation is initialized relative to some fixed epoch, and the value of the UDE epoch is known relative to some (other) fixed epoch time. The UDE initial value can be determined.

8.3 UDE value and UDE epoch known
The value of the UDE time is known at the start of the simulation, and the UDE epoch is known relative to some fixed-epoch time representation (i.e. a Standard Time). The values for some or all of the Standard Time representations can then be determined.

Rationale:

While conversion between these types is algorithmic, it can be time consuming and potentially prone to error. The provision of these capabilities allows the user to focus on getting the simulation dynamics correctly configured without worrying about whether some known time has been converted into a usable input correctly.

Verification:

Test time_3 on page 83 and test time_8 on page 88.

Requirement time_9: Ability to Overwrite Data Tables

Requirement:

The Time Representations Model requires data tables to tabulate the occurrence of leap seconds (UTC), and the time-dependent offsets (UT1). These data must be provided, along with the capability to overwrite the official data.

Rationale:

Data can only be provided for periods in which it is available. Neither data table has future predictive capabilities; users wishing to run simulations set in some future time may need the capability to assign a best-estimate to these values. Furthermore, any users wishing to test compatibility with releases of JEOD pre-dating the JEOD 2.0.0 release will need to turn off the update functionality of these tables, and overwrite with some particular value.

Verification:

Test time_9 on page 89.

Requirement time_10: Extensibility

Requirement:

The Time Representations Model shall provide the capability of extension to permit the user to develop clocks of specific interest that will operate within the time structure.

Rationale:

While every effort has been made to ensure that the time coverage is as complete as possible, we cannot imagine every potential application or situation in which the Time Representations Model will be utilized.

Verification:

Test time_10 on page 92.

2.1 Requirements Traceability

Table 2.1: Requirements Traceability

Requirement	**Inspection and Testing**
time_1 - Top-level Requirements	Insp. time_1
time_2 - Physical Time	Test time_1 Test time_2 Test time_11
time_3 - Necessary Clocks	Test time_3
time_4 - Necessary Representation Formats	Test time_6 Test time_7 Test time_8
time_5 - Time Initialization by Representation	Test time_4 Test time_5 Test time_8
time_6 - Time Initialization by Format	Test time_6 Test time_7 Test time_8
time_7 - Time Initialization Bypass	Test time_1
time_8 - Initialization of User-Defined-Epoch Times	Test time_3 Test time_8
time_9 - Ability to Overwrite Data Tables	Test time_9
time_10 - Extensibility	Test time_10

Chapter 3

Product Specification

3.1 Conceptual Design

This section describes the key concepts found in the Time Representations Model. For an architectural overview, see the *Reference Manual* [1].

3.1.1 Overview

The "second" is a well-defined quantity, counted to high precision by atomic clocks, and is incorporated into many physical constants, even into the definition of length. In a Newtonian universe, all atomic clocks would tick at the same rate; the fundamental unit of time may as well be the Earth-based atomic time (TAI) as any other atomic-time. In the vast majority of applications of JEOD (near-Earth simulations with millisecond precision), using TAI as the fundamental second is highly recommended, and 1 second of simulation time should equal 1 second of TAI. In rare exceptions, it may be desirable to use a more heliocentric central base-time (e.g. Barycentric Dynamic Time (TDB) or Ephemeris Time (not included in this release)), and some local time to track each vehicle independently.

In JEOD v5.1, there are many different clocks available, each of which has the following commonalities:

- A character-string name
- An initial value (value at sim-start)
- Identification (by name) of the time-type from which it is to be initialized
- Identification (by name) of the time-type from which it is to be updated
- A pointer to the converter function that will be used to update it
- A pointer to the Time Manager (*TimeManager*)
- A pointer to the Time Manager Initializer. (*TimeManagerInit*)

3.1.2 Simulator Time

In previous versions of JEOD, the simulator time (in a Trick simulation, this is *sys.exec.out.time*) was used as the basis for all dynamical processes. Everything progressed at the rate at which simulator time advanced. This is highly inflexible, and the JEOD v5.1 model has moved away from using the simulator time for anything more than an incremental counter. Simulator Time is NOT a part of the Time model.

3.1.3 Dynamic Time

To enhance flexibility, we have added an intermediary time to manipulate the simulation time to a clock appropriate for integrating the dynamics of the simulation. While simulation time always advances forward at a constant "rate", Dynamic Time (class: *TimeDyn*, object ID: *time.dyn*) can be made to speed up or slow down (thereby adjusting the temporal resolution without adjusting the calling frequencies), or even to run in reverse. TimeDyn can, in principle, be linked to any clock (e.g. TAI, UTC, UT1) with an appropriate converter, and thereby define that clock as the fundamental second. Because TAI is based on the SI second at Earth's gravitational potential, it is highly recommended that TAI be used, unless the user fully understands the implications of switching to a different fundamental clock.

This is really important, so it is worth repeating: it is this Dynamic Time, not the simulator time, that is to be used by the integrators. Simulator time has no physical significance whatsoever, it is purely used as a "placeholder" to monitor the computational progress of the simulation.

There are some other important issues associated with the switch from a *sim-time* based simulation to a *dyn-time* based simulation, due to the separation between simulation management and dynamic integration:

- Time resolution can now be adjusted trivially, while calls to functions are maintained at a constant ratio. For example, suppose that Function X is called every 3 seconds, and function Y called 3 times for every call to X (i.e. every second), and the user wishes higher resolution at a particular part of the simulation. It is sufficient to adjust the rate at which *dyn-time* advances. Adjusting by a factor of 10 would yield calls to function X every 0.3 seconds, and to function Y every 0.1 seconds. There is no need to adjust the job-cycle in the actual simulation.

- Previous time-resolution limitations associated with simulation management are no longer valid. Increased resolution can be achieved by setting *dyn-time* to run at a small fraction of the rate of *sim-time,* thereby increasing the available resolution without bound.

- A consequence of this feature is that the calling rate defined in the simulation will only be true if *dyn-time* and *sim-time* are running at the same rate. Again, setting the calling rate to be every 3 units in the simulation will be every 0.3 seconds if *dyn-time* runs at one-tenth the rate of *sim-time*.

- Adjusting the time-resolution away from a 1-1 ratio can also have unintended consequences in the operation of the simulation, particularly if the simulation is interfacing with some other

application. Adjusting the time-resolution could affect the rate at which data is exchanged, and only one side of the interface would be aware of the change.

Dynamic Time is automatically incorporated into every simulation as an inherent part of the Time Manager. Dynamic Time always starts the simulation at 0.0.

From *dyn_time,* the other times can be derived, although the default converters that come with JEOD v5.1mostly require that TAI be derived from *dyn-time* and everything else from TAI. All of the other times are therefore referred to as derived-times, which is then subdivided into Standard Times, and User-Defined-Epoch Times.

3.1.4 Derived Time

Depending on the simulation, it may or may not be necessary for additional time representations. If the purpose of a simulation is to observe some arbitrary dynamic event with no reference to any external configuration, then no additional times are needed. As soon as time-varying environmental factors are introduced, or the user wishes to use a clock that starts at some value other than 0.0, it becomes necessary to add additional capabilities. To convert between time representations, Time Converters are then needed. To manage all of the time representations into a coherent system, a Time Manager is also provided.

All additional times are derived from Dynamic Time (either directly, or via other derived times).

There are two fundamentally distinct types of these derived times – one that is well defined outside the simulation, and one that is not. The first is referred to as a Standard Time, the second as a User-Defined-Epoch Time (UDE). All derived times fall into one of these two categories, the concept of derived time becomes incorporated into both of these categories, and disappears as a stand-alone concept.

3.1.5 Standard Time (STD)

The concept of a Standard Time is that it is commonly accepted, with no ambiguity. Picking a particular value on a particular clock (e.g. noon on January 23, 2009, UTC) is well understood without additional context. Consequently, there can be only one clock running in a simulation for each of these times (e.g. 2 clocks running UTC would have to read exactly the same value, and would be entirely redundant).

The specific clocks all inherit from the base-class TimeStandard. The commonality between the various subclasses is extensive, therefore TimeStandard provides the vast majority of the structure and methods for those subclasses. Nevertheless, there is no known application that could utilize an instance of a TimeStandard (extension to add other clocks should be carried out by *inheriting* from TimeStandard, not by instantiating a TimeStandard). Consequently, the ability to instantiate a TimeStandard was deprecated in JEOD version 2.1, and removed in JEOD version 2.2.

3.1.6 User-Defined-Epoch Time (UDE)

The main distinction between a Standard Time and a User-Defined Time is that User-Defined Times are ambiguous without additional context. A commonly used example of a UDE (User Defined Epoch) is Mission Elapsed Time. Simply stating that a simulation started 2 hours after the mission started is not a very meaningful statement, unless we also have information on when the mission started. The *epoch* of UDEs (i.e. when their value was zero) must be defined before they make any sense at all; that epoch must ultimately be anchored with a Standard Time, or with the Dynamic Time. A secondary difference arises from the ambiguity of a UDE time; while Standard Times are restricted to one instance per time-type, the UDEs have no such restriction. A dozen or more different clocks could be running simultaneously, representing different time zones (ticking with UTC), or different mission-elapsed times for simulations involving multiple vehicles.

3.1.7 Epoch Definitions

An epoch is an instant in time; for our purposes, it is a particular time when the values of two or more clocks are known simultaneously. The most commonly used epoch in modern analysis is J2000, corresponding to noon TT on January 1, 2000. This is the default epoch used for all Standard Times (except GPS, which has its own well-defined epoch), although users are free to define their own. Indeed, users must define their own epoch for UDE times (as the name suggests), since these have no fixed epoch.

J2000 is defined with respect to Terrestrial Time; the value of other clocks is known at J2000, but it is not the same value as for Terrestrial Time. To tie clock values together, we use Truncated Julian Date. Truncated Julian Date is based on Julian Date, but with a more recent epoch; we use the NASA definition of Truncated Julian Date with a epoch of midnight on May 23/24, 1968. This value is specific to the clock under consideration (so UTC TJT=0 occurred at 00:00:00 May 24, 1968 UTC, which is a different instant in time from TT TJT = 0, or TAI TJT = 0).

It is important to differentiate between the two epoch concepts. The former is used to zero a time representation, while the latter is used to provide continuity between types. JEOD provides two variables – *seconds* and *days* – associated with the former interpretation for each time representation; these measure the elapsed time since that epoch. Each Standard Time (except GMST) also carries a variable *trunc_julian_time*, and a constant *tjt_at_epoch*; these are associated with the latter interpretation.

3.1.8 Time-keeping

A decimal counter-type clock was chosen as the basic time-keeping method in each time representation because it is more computationally efficient than conducting the constant verification for passing through boundaries that is inherent to a calendar-type clock. For all times, seconds-since-epoch, and days-since-epoch are both maintained. For most Standard Times, the epoch chosen is J2000 (12:00 noon TT on January 1, 2000). Additionally, for Standard Times, the NASA standard Truncated Julian Time is also maintained. The Truncated Julian representation was chosen for its improved accuracy over the Julian or Modified Julian representations, and for its versatility in comparing current values of standard clocks, and for converting to calendar-type representations.

The following example demonstrates the distinction between maintaining Truncated Julian Time and seconds since epoch (J2000).

> Terrestrial Time (TT) ticks in lockstep with atomic time (TAI), but they are offset by a constant value. The J2000 epoch occurred at the same instant for both, so the respective seconds-since-epoch values will be the same for both since they tick in lockstep. Conversely, the Truncated Julian Time "epochs" are clock-dependent, so they occurred at different instants. Since TT and TAI are offset from each other, their respective Truncated Julian Times values will always differ by a constant amount and their reported times will therefore differ, as is expected.

A calendar-based clock is also made available for most Standard Times; maintaining this is more demanding than the simple decimal-type clocks, so we have left this to the discretion of the user. It is an easy task to have the simulation routinely calculate the calendar representation, but the default is to omit the calendar representation. Similarly, for the UDE Times, there is a clock capability (day::hour::min::sec) that can be maintained if desired.

3.1.9 Initializing the Simulation

For maximum versatility, the code must be able to initialize the simulation given a time in any convenient format referencing any defined clock. The resulting consequence is that there is plenty of opportunity to *incorrectly* specify the initialization time. Users should take care to read through the initialization instructions found in the Analysis section of the User Guide (chapter 4).

3.1.10 Time Converters

Time Converters are needed to calculate the values of derived times (ultimately) from the Dynamic Time.

Converters between types are each in a separate class. All converters operate between two time-types, "type-a", and "type-b"; their classes are named, accordingly, TimeConverter_AAA_BBB. All converters have the following commonalities:

- The name of type-a
- The name of type-b
- Identification of the available functionality of the converter; each converter has two directions available (a-to-b, and b-to-a), and two potential operation times (initialization, and runtime). Each converter class has four flags, one for each combination. to indicate how the converter may be used.
- A numerical difference between type-a and type-b. In some cases, this is a constant, in others it is calculated, in others it is obtained from data tables.

3.1.11 Time Manager

The Time Manager controls all of the time representations.

The initialization of the class TimeManager is handled by the class TimeManagerInit.

The Time Manager contains a list of all the time representations that are to be updated during the simulation.

The Time Manager first updates the Dynamic Time, based on the Simulator Time. Each of the derived-time representations then update from their respective parent time, starting from *dyn-time* and propagating down the time-update-tree as needed.

The TimeManager maintains two lists:

- All registered JeodBaseTime-derived time instances.
  - This list is created at initialization, and after which it remains constant throughout the simulation.
  - At the end of the initialization cycle, all times and their conversions should be identified. At this point, the list of time objects is sorted in the order that their update must take place.
  - Each time instance knows where it is located in that list.
  - The TimeManager has a lookup function to find the location on that list of any time object by name.
  - The Time Manager keeps all of these types updated throughout the simulation.
- All registered Time Converters.

3.1.12 Time Update Tree

This is a tree built at initialization that is stored in the TimeManager class. Coming from TimeDyn, the tree branches to any time classes scheduled to be updated directly from DynTime. Then from each of those to the next layer down, etc. For each branch, a Time Converter is needed. If the tree cannot be built at initialization (due to insufficient Time Converters), the simulation will fail immediately. The hierarchical list of time types maintained by the TimeManager is derived from this tree, then knowledge of the tree is discarded.

3.1.13 Time Manager Initialization

The TimeManagerInitializer performs the following tasks:

- Creates a tree-like structure for initialization and updating hierarchy
  - Includes auto-finding paths, and user-entered override capabilities
  - Ensures that there are no circular dependencies or orphans

- Identifies which converter functions are required for navigating the tree of time-types
- Initializes all of the time representations.

3.2 Mathematical Formulations

The mathematics of the Time Representations Model model is remarkably simple, with little beyond basic addition and subtraction of time increments and offsets. The major exceptions are in the functions *convert_from_calendar* (on page 44) and *calculate_calendar_values* (on page 41), which are described in the *Functional Design* (on page 23) section of the Product Specification.

3.3 Detailed Design

This section is divided into 4 parts:

- A process flow-through description (Process Architecture) of the sequence in which the various functions are called, and the interaction between the objects.
- A complete list and description of all methods that comprise the Time Representations Model.
- A description of the external data files used by the objects.
- A description of the extension capability of the model.

Further, the *Reference Manual* [1] contains a structural overview of the Time Representations Model.

3.3.1 Process Architecture

This section describes the flow from function to function. The operation of each function is described under its object in the section *Functional Design* (on page 23).

3.3.1.a Overview of calling sequence (initialization)

1. **TimeManager::register_time** or **TimeManager::register_time_named**

   Stores the address of the manager in each time-type. Adds the type to the manager's list of time-types, which is stored as a vector of pointers to each type. Then stores the index (position in the vector) of each type in each type.

2. **TimeManager::register_converter**

   Adds the converter to the TimeManager's list of converters.

3. **TimeManager::initialize**

   Registers and initializes the dynamic time. Records the number of time-types.

   (a) **TimeManagerInit::initialize_manager**

   Sequence of calls

   i. **TimeManagerInit::initialize**
   Records the dyn_time index, and initializer index.
   Allocates memory.
   Initializes arrays.

   A. **TimeManager::time_lookup**
   Look up the initializer type by name.

   B. **TimeManagerInit::verify_times_setup**
   Sanity checks.

   ii. **TimeManagerInit::populate_converter_registry** [1]
   Stores pointers in the appropriate locations (a-to-b and b-to-a) for each of the converters stored by the Time Manager.
   Stores +1 / 0 / -1 in each location of the direction tables (+1 represents a-to-b, 0 indicates that the converter may not be used, -1 represents b-to-a).

   (b) **TimeManagerInit::verify_converter_setup**

   Comprises sanity checks for where potential conflicts could occur between different converter classes.

   i. **TimeManager::time_lookup** *(optional)*
   Look for the TAI, UTC, and UT1 time instances to verify correct setup.

[1] **Converter_registry**

This is a registry that allows quick verification and lookup of functions that convert from one time type to another.

There are 2 converter registries, one for initialization, and one for runtime. Both take advantage of the C++ vector, but the initialization registry is much larger than the runtime registry.

The first registry is used during the initialization process and contains 3 vectors that represent n x n arrays, where n is the number of time representations. The converter from time i to type j is associated with element *(ni + j)* of the vectors.

A. One vector *(converter_class_ptr)* contains pointers to the associated converter class.

B. A second vector *(init_converter_dir_table)* contains a value -1, 0, or 1. The *init* in the name indicates that this shows the direction in which the converter should be used during *initialization*. The number indicates the direction (-1 reverse; 0 not available; 1 forward).

C. A third vector *(update_converter_dir_table)* does the same for converters to be used during the simulation.

The second registry is used at runtime and contains only 2 vectors, each with only n elements:

A. update_converter_ptr. Element i contains the pointer to the converter class that is used to update time type i.

B. update_converter_dir. Element i contains -1, 0 or 1 indicating the direction in which the converter should be used.

(c) **TimeManagerInit::create_init_tree** [2]

Multi-pass function that repetitively calls add_type_initialize on all time-types that are not in the tree until everything is in the tree, or nothing else can be added.

i. **Time::add_type_initialize**
Uses the user-specified "initialize_from_ name" values to place type. Where names are missing, type is placed in the tree based on the availability of converter functions.

A. **TimeManager::time_lookup** *(optional)*
Look up the time types by name.

(d) **TimeManagerInit::initialize_time_types** [3]

Sequence of calls

i. **Time***::initialize_initializer_time** [4] *(optional)*
Initialize the time specified by the user as the "initializer" (this is at the head of the initialization tree).

A. **Time***::convert_to_julian** *(optional)*
If initialization data is in calendar format, this function converts it to Truncated Julian format.

ii. **Time::initialize_from_parent**
Works down through the tree, initializing each time type from its parent in the initialization tree.

A. **TimeConverter::initialize** [5] *(optional)*
Initializes any converter functions that are needed in the process.

[2] **Initialization tree**
Each time type has a record of how it gets initialized, and one time type is designated as the "initializer". The *create_init_tree* function builds a tree starting with the initializer, and working down to those types that are initialized based on the value of the initializer, then those that are based on those that are based on the initializer, etc. If the user omits *an initialize_from* specification for a particular type, this function will attempt to identify a suitable initialization path from the converter registry. Structures that leave types isolated from the initializer, or produce loops, will cause a termination.

[3] **Initialize time types**
The function *initialize_time_types* first calls *initialize_initializer_time* to set the values in the time-type that resides at the top of the initialization tree, then calls *initialize_from_parent* on each un-initialized time type. *initialize_from_parent* is a recursive function which will call itself on the parent time-type if the parent is not yet initialized.

[4] **Initialize initializer**
If dyn_time is the only time class, it is the initializer by default, and initializes to zero. Otherwise, the user-defined initializer time is assigned its initial values.

[5] **Initialize the converters**
To set the time from the parent, the converter functions must be available. Some of these may require some initialization (e.g. the TAI to UTC converter requires finding the appropriate values from data tables). If the converter from parent to child has not been initialized, that is completed before the child's time is set as a call from *initialize_from_parent*.

If the converter has not been registered, the code will terminate. This is redundant; if the converter had not been registered, it could not have been used in building the initialization tree.

(e) **TimeManagerInit::create_update_tree** [6]

Sequence of calls

i. **Time::add_type_update** [7]

Repeats the tree-building exercise carried out in create_init_tree, but uses the update side of the converter functionality and starts with *dyn-time* instead of "initializer".

A. **TimeManager::time_lookup** *(optional)*
Look up the time types by name.

B. **TimeManager::organize_update_list**
Generates the hierarchical list of time types in the order in which they must be updated. This is stored in TimeManager.

C. **TimeConverter::initialize** *(optional)*
Called if the necessary time converter has not been called.

4. **TimeManager::update** [8]

Run a full update on all times with *dyn-time* = 0.0.

[6] **Create the update tree**

The update tree is built in much the same way as the initialization tree. The top level of the update tree must be TimeDyn, which is the time used for integration and is derived directly from the simulation engine. While the initialization process was distributed over many functions, the assignment of the update process is more compact, although the same checks are made that each time type can "see" TimeDyn, and that there are no loops.

[7] **Adding types to the update list**

Recursive function. Ensures that the parent type is in the tree, or calls itself on the parent if not, then records the appropriate converter functions in the runtime converter registry using *organize_update_list*

[8] **Update all times**

Although all times have been set to their starting values, there may be some small numerical differences that occur when comparing the converters as used during initialization to those used during runtime. These potential differences will have negligible effect on an absolute time reference, but may be noticeable with relative values. Thus, starting with one structure and switching to another should be avoided. The initial times at the start of the simulation will be propagated down from TimeDyn in exactly the same way that they will be during the run.

3.3.1.b Overview of calling sequence (runtime)

1. **TimeManager::update**

   Sequence of calls:

   (a) **TimeDyn::update**

   Updates Dynamic Time first.

   (b) **Time***::update**

   Call to the update function in each of the time classes.

   i. **TimeConverter::convert_a_to_b / convert_b_to_a** [9]

   Run the appropriate converter function.

   (c) **TimeDyn::update_offset**

   If the scale factor,m, has changed on dynamic time, this function adjusts the offset, c, of dynamic time, d, from simulation time, s, so that $d = ms + c$.

2. **(TimeStandard::calendar_update)** *(optional)* Called on select derived times when a calendar-based output is desired.

[9] **Converter function**
The time class knows which converter class to use, and in which direction to use it.

3.3.2 Functional Design

See the Reference Manual [1] for a summary of member data and member methods for all classes. This section describes the functional operation of the methods in each class.

Objects are presented in alphabetical order. Methods are presented in alphabetical order within the object in which they are located. Methods that are inherited from a parent class are described in that parent class.

1. **Class: JeodBaseTime** **Inheritance:** Base-class

   (a) **add_type_initialize**
   Specific to each time-type. If not specified for a time-type, causes a termination. This function is specified in classes *TimeUDE* (on page 47) and *TimeStandard* (on page 40), but not in *TimeDyn*.

   (b) **add_type_update**
   Recursively adds elements to the update tree. If the "parent" to a time-type is defined but not already in the tree, it calls itself on the "parent" then returns to adding the "child" type. If the "parent" is not defined it searches for a suitable "parent" from the types already in the tree. If that search is successful, it adds the "child" to the tree, otherwise it returns without change.

   (c) **initialize_from_parent**
   Specific to each time-type. If not specified for a time-type, this method causes a termination. This method is specified in classes *TimeUDE* (on page 47) and *TimeStandard* (on page 40), but not in *TimeDyn*.

   (d) **initialize_initializer_time**
   Pure-virtual function.

   (e) **must_be_singleton**
   Returns true by default, indicating that there can be only one type of a particular object.

   (f) **set_time_by_days**
   Takes an input value, and sets the value *days* to that value, and the value *seconds* to 86,400 times that value.

   (g) **set_time_by_seconds**
   Takes an input value, and sets the value *seconds* to that value, and the value *days* to 1/86,400 of that value.

   (h) **update**
   Calls the appropriate converter to update the time from its parent.

2. **Class: TimeConverter** **Inheritance:** Base-class

   This is the abstract class that describes the basic structure needed by all converters. The different subclasses each describe the conversion process between a specified pair of time-types. Only a small subset of potential pairings are included in the release of JEOD v5.1, although that small subset does provide complete coverage of all the clock representations.

Table 3.1: Availability of Converter Functions

This table shows which converter functions are available in the standard release.
1 – available for initialization only
2 – available for updates only
3 – available for initialization and updates.
* – by inheritance.

To: / From:	Dyn	GMST	GPS	MET	STD	TAI	TDB	TT	UDE	UT1	UTC
Dyn				3*		2	2	3			
GMST											
GPS						3					
MET					3*						
STD				3*					3		
TAI			3				3	3		3	3
TDB					3						
TT						3					
UDE					3						
UT2		3				3					
UTC						1					

(a) **can_convert**

Specific to each converter type (not to each instance). Extended classes may combine these options using bitwise "OR" when assigning in the constructor.

To check whether the converter can accept any combination of directions, the user may utilize the can_convert() function

NO_DIRECTION : null, no direction specified

A_TO_B_INIT : able to initialize the second time from the first

B_TO_A_INIT : able to initialize the first time from the second

A_TO_B_UPDATE : able to update the second time from the first

B_TO_A_UPDATE : able to update the first time from the second

A_TO_B : always able to convert the second time from the first

B_TO_A : always able to convert the first time from the second

ANY_DIRECTION : always able to convert the two times

(b) **initialize**

Pure-virtual function.

Each subclass must define its own version of this function, but they all have some similarities.

Each time converter has the potential to contain two functions, a converter from *aaa* to *bbb*, and a converter from *bbb* to *aaa*.

The initialize function receives 3 arguments – a pointer to a time-type labeled as *parent,*

a pointer to a time-type labeled as *child,* and an integer ±1 that identifies whether *aaa* is associated with *parent,* or *child.*

The converter is initialized for conversion either from *aaa* to *bbb* or for conversion from *bbb* to *aaa.* However, the initialization of the converter then allows the conversion in both directions, if that functionality exists. This is best seen in an example:

> Suppose that when building the tree, the TimeManager identifies that time-type *xxx* is calculated from time-type *yyy.* It identifies that a time-converter, *TimeConverter_xxx_yyy* has been registered for this type of conversion, and that this converter has not been initialized. It calls *TimeConverter_xxx_yyy:: initialize(yyy_ptr, xxx_ptr, -1)* because *yyy* is the *parent* and *xxx* the *child* for the intended conversion. If the converter is not able to convert from *yyy* to *xxx*, the tree that has been defined is invalid, and the code will terminate here. This initialization will then also allow conversions from *xxx* to *yyy* if the *convert_a_to_b* function is defined.

The *verify_setup* function is called with the same arguments. In order for the verification to proceed, one of the two types – *child* or *parent* – must be already initialized. This type is called the *master.* In most cases, the *parent* type becomes the *master* type for the verification, but there are exceptions. See *Time_Converter_dyn_tai::initialize* (on the current page) for discussion.

Next, the *parent_ptr* and *child_ptr* are cast into their respective time-types, and a check made that they are of the time-type that they are supposed to be.

Finally, the initial value *a_to_b_offset* may be calculated, which can (in some cases) be used in the *convert_a_to_b* and *convert_b_to_a* functions. In some converters, this initial value remains unchanged throughout the simulation.

(c) **verify_setup**

This function is used by each of the *TimeConverter_xxx_yyy* classes to verify that sufficient data exists to allow the converter to be initialized. The following criteria must all be met or the code will terminate:

i. Of the two time-types associated with the converter, one (identified as the *master* in this function) must already be initialized.

ii. Both time-types must have known memory addresses.

iii. A direction *(1 or -1, indicating xxx is master or yyy is master respectively)* must have been defined.

(d) **verify_table_lookup_ends**

Has no functionality at this level, simply returns to the calling routine.

3. **Class: TimeConverter_Dyn_TAI** **Inheritance:** *TimeConverter* (on page 23)

(a) **initialize**

See the base-class version of *initialize* (on the preceding page) for overall description.

Because *TAI* has an absolute reference, and *dyn-time* does not, the initial value of *TAI* must be known to initialize the offset between them. Therefore, the *master* type must be *TAI* independent of which is the *parent.* The *initialize* function could be called with the intention of confirming *TAI* to *Dyn* conversion capability or *Dyn* to *TAI* conversion

capability, but either way, the *master* must be *TAI*. Calling *verify_setup* with *master* = *Dyn* would fail to provide the protection that this function is intended to provide.

Once initialized, the offset between *Dyn* and *TAI* is constant throughout the simulation.

(b) **can_convert**

valid_directions: A_TO_B_UPDATE

For runtime only. The offset between *time_dyn* and *time_tai* is constant throughout a simulation. Knowledge of this offset and of the value *dyn_time* allows for a trivial calculation of *time_tai*.

4. **Class: TimeConverter_Dyn_TDB** **Inheritance:** *TimeConverter* (on page 23)

(a) **initialize**

See the base-class version of *initialize* (on page 24) for overall description.

Because *TDB* has an absolute reference, and *dyn-time* does not, the initial value of *TDB* must be known to initialize the offset between them. Therefore, the *master* type must be *TDB*independent of which is the *parent*. The *initialize* function could be called with the intention of confirming *TDB* to *Dyn* conversion capability or *Dyn* to *TDB* conversion capability, but either way, the *master* must be *TDB*. Calling *verify_setup* with *master* = *Dyn* would fail to provide the protection that this function is intended to provide.

Once initialized, the offset between *Dyn* and *TDB* is constant throughout the simulation.

(b) **can_convert**

valid_directions: A_TO_B

The offset between *time_dyn* and *time_tdb* is constant throughout a simulation. Knowledge of this offset and of the value *dyn_time* allows for a trivial calculation of *time_tdb*.

5. **Class: TimeConverter_Dyn_UDE** **Inheritance:** *TimeConverter* (on page 23)

There can be multiple instances of this class.

(a) **initialize**

See the base-class version of *initialize* (on page 24) for overall description.

This function is similar to the *Dynamic Time to TAI initializer* (on the preceding page), in that the UDE time may have a defined initial value known only to TimeUDE. The *master* type must be *UDE,* even though *Dyn* is always going to be the *parent*. The *initialize* function could be called with the intention of confirming *Dyn* to *UDE* conversion capability, but the *master* must still be *UDE*. Calling *verify_setup* with *master* = *Dyn* would fail to provide the protection that this function is intended to provide.

Once initialized, the offset between *Dyn* and *UDE* is typically constant throughout the simulation. The exception to this is the case of a *MET* (subclass of *UDE*), which allows for hold-points in the simulation. In this case, the offset will be adjusted accordingly during the simulation.

(b) **can_convert**

valid_directions: A_TO_B

The offset between *time_dyn* and *time_ude* is well known throughout a simulation. *time_ude* can easily be calculated with knowledge of this offset and of the value *dyn_time*.

6. **Class: TimeConverter_STD_UDE** **Inheritance:** *TimeConverter* (on page 23)

   There can be multiple instances of this class.

   (a) **can_convert**
   valid_directions: ANY_DIRECTION
   For run-time and initialization.

   (b) **initialize**
   See the base-class *initialize* (on page 24) method for overall description.

   (c) **reset_a_to_b_offset**
   In the event that a hold was placed on the UDE time, the offset between the two times will change. This function simply recalculates the offset by differencing the two times once the hold has been removed.

7. **Class: TimeConverter_TAI_GPS** **Inheritance:** *TimeConverter* (on page 23)

   The difference between the Truncated Julian Time values of *GPS* epoch time and *TAI* epoch time is taken (these default to midnight UTC on January 5/6, 1980, and J2000 respectively). This gives a difference in days, which is used to calculate the offset. The offset remains constant though the simulation.

   (a) **can_convert**
   valid_directions: ANY_DIRECTION
   For run-time and initialization.

   (b) **initialize**
   See the base-class *initialize* (on page 24) method for overall description.

8. **Class: TimeConverter_TAI_TDB** **Inheritance:** *TimeConverter* (on page 23)

   The offset between these two times can be calculated from an analytic function to a very high degree of precision each time it is needed. It is not given an initial value.

   (a) **can_convert**
   valid_directions: ANY_DIRECTION
   For run-time and initialization. Converter is straightforward, TDB is an analytic function of TAI. A generalized version of the code is provided below.

```
void TimeConverter_XXX_YYY::convert_a_to_b (void)
{
set_a_to_b_offset();
yyy_ptr->set_time_by_seconds (xxx_ptr->seconds + a_to_b_offset -
      a_to_b_offset_epoch);
return;
}

void TimeConverter_XXX_YYY::convert_b_to_a (void)
{
xxx_ptr->set_time_by_seconds(prev_xxx_seconds + (yyy_ptr->seconds - prev_yyy_seconds)
```

```
while(true) {
nSteps++;
nIter++;
set_a_to_b_offset();
double dXXX = (yyy_ptr->seconds - xxx_ptr->seconds)
    - (a_to_b_offse - a_to_b_offset_epoch);
xxx_ptr->set_time_by_seconds(xxx_ptr->seconds + dXXX);

if (nSteps > 5 || std::abs(dXXX/xxx_ptr->seconds) < std::pow(10,-15)) {
break;
prev_yyy_seconds = yyy_ptr->seconds;
prev_xxx_seconds = xxx_ptr->seconds;
}
}
}
```

(b) **initialize**

See the base-class *initialize* (on page 24) method for overall description.

9. **Class: TimeConverter_TAI_TT** **Inheritance:** *TimeConverter* (on page 23)

TT is derived from the old ephemeris time which, at the time of its definition epoch, happened to have an offset of 32.184 seconds, or 0.0003725 days. This offset is constant.

(a) **can_convert**

valid_directions: ANY_DIRECTION

(b) **initialize**

See the base-class *initialize* (on page 24) method for overall description.

10. **Class: TimeConverter_TAI_UT1** **Inheritance:** *TimeConverter* (on page 23)

The techniques used in the TAI to UT1 converter are similar to those in *Time_Converter_TAI_UTC* (on page 30). The main difference is that both UT1 and TAI are continuous, and the offset between them is calculated by an interpolation between the boundary points of the time interval in the *tai_to_ut1* table. This avoids the problem of handling discontinuities that are discussed in the *TAI to UTC converter* (on page 30), but introduces another concern over interpretation of data.

The lookup times in the TAI_to_UT1 tables can be interpreted as being either TAI or UT1, but that interpretation must be consistent. The arbitrary interpretation is permitted because they differ by so little, and that difference changes so slowly. That difference is typically $TAI - UT1 \sim 30\ s$, it changes by $O(10^{-3})\ s \cdot day^{-1}$ and is calculated out to $O(10^{-4})\ s$. Making a wrong interpretation introduces an error that is at most the product of the gradient with the maximum difference, equating to $O(10^{-7})\ s$, which is well within the precision of the tables themselves. Because UT1 is typically updated from TAI, it is assumed that these tables represent the lookup times as a TAI representation.

Nevertheless, if the interpretation is inconsistent, it would be possible to use TAI to find UT1, then use that value to convert back to TAI, ad infinitum, progressing time artificially.

A particular example is the situation in which TAI is initialized based on a UT1 time, then UT1 updated from TAI; because UT1 is calculated from TAI there would first be a UT1-based lookup to generate TAI, then a runtime TAI-based lookup to generate UT1. Not accounting for the difference would result in the simulation starting at a time $O(10^{-7})$ s different to when it was intended to start. This leads to a difference in the way the two converters function.

(a) **can_convert**

valid_directions: ANY_DIRECTION

Since UT1 is calculated from TAI by the linear interpolation of points in the tables, we have:

$$UT1 = TAI + \left(offset_{prev} + (TAI - \ mathitwhen_{prev})grad\right) \tag{3.1}$$

where

$offset_{prev}$ is the most recent entry in the TAI-to-UT1 conversion table, giving the value UT1 - TAI,

$when_{prev}$ is the TAI time corresponding to $offset_{prev}$,

and

$$grad = \frac{offset_{next} - offset_{prev}}{when_{next} - when_{prev}}$$

While it is not appropriate to use the same linear interpolation to obtain TAI, this expression can be rearranged to give a comparable

$$TAI = UT1 - \left(\frac{offset_{prev} + (UT1 - when_{prev})\, grad}{1 + grad}\right)$$

(Note that $when_{prev}$ is still TAI time, with a corresponding UT1 time of $(when_{prev} + offset_{prev})$)

In most situations, this is sufficient, but a problem occurs when the gradient calculated based on the UT1 time input is different to that for the TAI time output, i.e. when the TAI and UT1 times are in different "boxes" in the tables. The gradient in the above equation should be associated with the TAI time, but it is first identified from the UT1 time. To circumvent this problem, the value that the UT1 time is compared against is not the interval boundary (known in TAI time), but the sum of the interval boundary and the offset at that boundary (UT1 time at the boundary). Because UT1 is continuous at the boundaries of each "box", this eliminates the inconsistency.

(b) **initialize**

See the base-class *initialize* (on page 24) method for overall description.

(c) **initialize_tai_to_ut1**

This function is used to initialize the converter from TAI to UT1, and set up the tables for future updates.

It has similarities to *initialize_leap_seconds* (on page 31) in setting up the table. Unlike the TAI to UTC conversion function, which has discontinuities, the TAI to UT1 conversion is a smooth function and requires that the previous and next data values be considered, and the slope between them calculated in order to interpolate between points in the tables.

(d) **verify_table_lookup_ends**

This function is necessary for those converter classes that utilize lookup tables. If a simulation time passes beyond the domain of a lookup table, the value of the offset is set to be the last known value, and a flag is set to indicate that further (time-consuming) calls to the table are not necessary. This flag must be reset if the simulation reverses time; this function resets that flag.

11. **Class: TimeConverter_TAI_UTC** **Inheritance:** *TimeConverter* (on page 23)

The conversion between TAI and UTC is based on lookup tables that provide the offset as a function of time. The times in those tables are UTC times, showing when leap seconds were introduced. In the cases that the offset value is artificially forced to remain constant (*true_utc = false,* for comparison to JEOD 1.x.x) or the current time is outside the range of the data table, the offset value is constant and the conversion trivial in either direction. It is recommended that TAI be updated from Dynamic Time, and UTC updated from TAI; this makes TAI smooth and UTC discontinuous, with repetition at leap seconds (hence, there is no Dynamic Time to UTC converter provided).

(a) **can_convert**

valid_directions: A_TO_B and B_TO_A_INIT

For converting from TAI to UTC, UTC is first calculated from the previously known offset. This value is then used for comparing against the boundaries. If the time has passed through a leap second value, the UTC representation is recalculated based on the corrected number of leap seconds, and tested again.

The conversion from TAI to UTC has issues at leap seconds. Because UTC is calculated from the previously known offset, it will pass through midnight before the new offset is identified. A typical sequence of UTC times, updated every 0.2 seconds may be (day:hour:minute:second)

[0:23:59:59.8 , 0:23:59:60.0 = 1:00:00:00.0 => 1:00:00:(-1.0) = 0:23:59:59.0 , 0:23:59:59.2 etc.]

Triggering the leap second at midnight causes UTC to momentarily progress one day, then back up again into the previous day, which is not a realistic representation. Of course, the alternative of triggering it at midnight + 1 second is worse since it causes UTC to start the next day 1 second early. This is the best solution available.

For converting UTC to TAI at initialization, the UTC value is compared to the recorded boundaries of the current calibration interval (*prev_when* and *next_when*). If it is outside the boundary, then the values of the boundaries and the value of the offset between UTC and TAI are adjusted accordingly. There is also verification that the current time has not passed the end of the calibrated data.

The conversion function from UTC to TAI is fundamentally flawed at the times where leap seconds are added; UTC normally ticks 23:59:59 to 23:59:60 = 00:00:00, except at leap seconds when it ticks 23:59:59 to 23:59:60 to 00:00:00. However, the Julian representation reads 23:59:60 and 00:00:00 as being the same time. The result is that the UTC value in the new day ticks from 0 to 1 second, jumps back to 0, and restarts, causing there to be two TAI values for any given UTC time in that interval. The code does not have the capacity to uniquely represent times in the interval $t \in$*[23:59:60,*

23:59:61) UTC. The problem really only shows itself at initialization (unless the user chooses to declare this converter valid at runtime). Even though 23:59:60.5 UTC may exist in reality, the code will interpret that value as being 1 second later, 00:00:00.5. Users should therefore avoid initializing a simulation using a UTC value during a leap second interval.

(b) **initialize**

See the base-class *initialize* (on page 24) method for overall description.

(c) **initialize_leap_seconds**

This function is used to initialize the UTC to TAI converter, in conjunction with *initialize* above.

The conversion between TAI and UTC utilizes a lookup table, which tabulates when leap seconds were added. The times used in the table are in a UTC representation, and the result is the difference between TAI and UTC.

Initially, a test is made whether the current UTC time is covered by the tables. If not, the end-value from the table is used; there is no attempt made to extrapolate the occurence of leap seconds either into the future, or before 1972.

In the event that the UTC time is not within the table range and never will be, a flag *off_table_end* is set to true; this has the effect of removing the lookup from the subsequent process.

If the current UTC time is within the boundaries of the table, then the number of leap seconds appropriate for that time is identified and recorded as (negative) *a_to_b_offset.*

(d) **verify_table_lookup_ends**

This function is necessary for those converter classes that utilize lookup tables. If a simulation time passes beyond the domain of a lookup table, the value of the offset is set to be the last known value, and a flag is set to indicate that further (time-consuming) calls to the table are not necessary. This flag must be reset if the simulation reverses time; this function resets that flag.

12. **Class: TimeConverter_UT1_GMST** **Inheritance:** *TimeConverter* (on page 23)

(a) **can_convert**

valid_directions: A_TO_B

For run-time and initialization, this algorithm is based on the Astronomical Almanac [4], with some simplification to account for our ability to count in days since J2000. The algorithm in the Astronomical Almanac uses days since noon on January 1, 2000 *UT1*. We carry days since J2000 (noon of January 1, 2000, *TT*), which is slightly different. To reconcile this difference, we first subtract off the difference between the two values, which corresponds to a little over a minute, or 0.00738762 days.

$$d = days_{UT1} - 0.000738762$$

(where $days_{UT1}$ is days of *UT1* time since J2000).

Then, the Astronomical Almanac algorithm can be expressed as:

$$days_{GMST} = 0.7790572733 + 1.002737909350795d + 8.0775E-16d^2 - 1.5E-24d^3 \quad (3.2)$$

and this value used to set *GMST* values by days.

(b) **initialize**

See the base-class version of *initialize* (on page 24) for overall description.

13. **Class: TimeDyn** **Inheritance:** *JeodBaseTime* (on page 23)

(a) **initialize_initializer_time**

Initializes the simulation only if there are no Standard Times present in the simulation.

(b) **update**

Updates the Dynamic Time value directly from the Simulator Time, using $t_d = \alpha t_s + \delta$, where:

- t_d represents Dynamic Time (seconds)
- t_s represents the Simulator Time (seconds)
- α represents the scale factor between the rate at which Dynamic Time advances, and the rate at which the Simulator Time/counter advances. This is user-defined, and defaults to 1.0.
- δ represents the offset between the Simulator Time and the Dynamic Time. This value is auto-generated, and defaults to 0.0. See *update_offset* (on the current page) for description of this value.

(c) **update_offset**

The Dynamic Time advances linearly with the Simulator Time, at some user-defined rate (default = 1.0), and with some *offset* value that represents the value of Dynamic Time when Simulator Time = 0.0. At the start of all simulations, both Dynamic Time and Simulator Time are initialized at 0.0, and the offset is therefore also 0.0. However, if the rate at which time advances changes mid-simulation, the offset must be recalculated.

Dynamic Time contains some public and private elements relating to the rate and direction in which time advances. The public element, *scale_factor*, can be changed in the input file but is not used directly anywhere in the code. Conversely, the private element, *ref_scale,* is actually used in updating Dynamic Time, and can only be changed here.

Only if the private and public elements disagree has a change been made to the public values. If the public values have changed, the private values must be updated, and the offset between Simulator Time and Dynamic Time recalculated.

In the following illustration, Dynamic Time and Simulator Time advance together. *Offset* retains its default value of 0.0. At some point (A in figure) Dynamic Time is slowed for enhanced resolution, and *offset* is calculated (to a). As long as the rate remains constant, *offset* will also remain constant. At some time later (B in figure), the rate of Dynamic Time with respect to simulator time is reverted to its original value. At this point, *offset* does not revert back to 0.0; instead, it is recalculated to give the value represented by b in the figure.

This function serves dual purpose. First, it tests whether the rate (or *scale_factor*) has changed, and recalculates *offset* only if it has. Secondly, it returns a true/false flag indicating the result of that test; this flag can then be used in other routines as necessary without further testing (e.g. integration algorithms that retain a history will have to be reset every time *scale_factor* is changed).

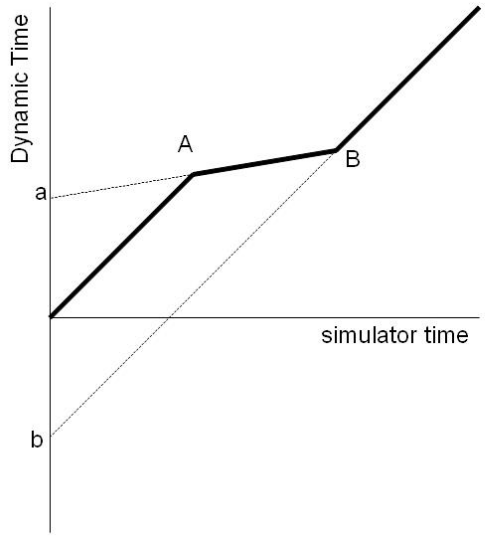

Figure 3.1: Illustration of how Dynamic Time can vary with Simulator Time, and the resulting effect on the offset between them.

14. **Class: TimeEnum** **Inheritance:** Base-class

    This class contains no methods, just an enumeration.

    (a) Julian.

    (b) Modified Julian.

    (c) Truncated Julian.

    (d) Calendar.

    (e) Clock.

    (f) Days since epoch.

    (g) Seconds since epoch.

15. **Class: TimeGMST** **Inheritance:** *TimeStandard* (on page 40)

    (a) **set_time_by_trunc_julian**

    Even though GMST is technically a Standard Time, Truncated Julian Time has no meaningful interpretation, since it is based on a synodic clock. This function overrides the one inherited from *TimeStandard* (on page 40) by causing a termination if anything attempts to use Truncated Julian Time in GMST.

    (b) **calculate_calendar_values**

    Even though GMST is technically a Standard Time, the calendar that is an element of Standard Time has no meaningful interpretation, since it is based on a synodic clock. This function overrides the one inherited from *TimeStandard* (on page 40) by causing a termination if anything attempts to use a calendar in GMST.

16. **Class: TimeGPS** **Inheritance:** *TimeStandard* (on page 40)

GPS time is somewhat unusual. It fits the definition of a Standard Time (it is well defined), but does not maintain the same counters as a typical clock. Instead of days, months, etc., it uses weeks and seconds of week since epoch (there is also a *rollover* count, used because the number of weeks since epoch has now exceeded its counting capacity of 1,024 weeks).

(a) **calculate_calendar_values**

The conventional concept of calendar makes no sense in the GPS timekeeping system. All GPS values are maintained all of the time, and any call to this function will produce a code termination.

(b) **set_time_by_days**

Converts number of days since epoch into weeks and seconds of week since epoch.

(c) **set_time_by_seconds**

Converts number of seconds since epoch into weeks and seconds of week since epoch.

(d) **set_time_by_trunc_julian**

The epoch is hard-coded in Truncated Julian Time, so it is trivial to convert any given Truncated Julian Time and change that into days or seconds since epoch.

17. **Class: TimeManager** **Inheritance:** Base-class

(a) **get_converter_ptr**

As each converter is registered, it is stored in a vector and is assigned an index value equal to its position within that vector. This function returns a pointer to any particular converter given its index in that vector.

(b) **get_time_change_flag**

The *time-change-flag* is triggered if the Dynamic Time scale factor is changed (see *update_offset* (on page 32)). This function simply returns the boolean indicating whether the scale factor changed since the last time step.

(c) **get_time_ptr**

There are two instances of this function, one takes a name, and the other takes an index. When a time representation is registered with the Time Manager, a pointer to it is stored in a vector. All time representations also have a unique name. This function returns the pointer to the time representation using either value as a reference.

(d) **initialize**

Registers and initializes the Dynamic Time.

Counts the number of time-types registered from the S_define.

Calls *TimeManagerInit::initialize_manager*.

Runs an update on all time-types at Dynamic Time = 0.0.

(e) **register_converter**

Called from the S_define level, this function simply appends a pointer to each converter onto the *time_converters_ptrs* vector. This function is used when the converter is between two specific time representations (e.g. TAI and UTC).

(f) **register_generic_converter**

Called from the S_define level, this function simply appends a pointer to each converter onto the *time_converters_ptrs* vector. This function is used when one or both of the two time representations is not a Standard Time (i.e. is a UDE or MET). This function requires naming the non-Standard representation to identify which representation is intended.

(g) **register_time_named**

This function is generally called from the S_define for a non-Standard Time representation, using the representation's name as an identifier. It adds each of these representations into the registry of time-types.

(h) **register_time**

This function is called from the S_define for each Standard Time representation (Standard Times are already named). It adds each of these representations into the registry of time-types.

(i) **time_standards_exist**

Returns a boolean value, indicating whether there are any time representations that fall into the *TimeStandard* classification.

(j) **time_lookup**

Each time-type has its own class, and is registered with the Time Manager, with a pointer to the class in a vector registry. The location of the time-type in the registry is the *index* of that time-type. The index is also stored in each class, as is the name of the class.

The index of a time-type is the point of reference throughout the simulation, and the address of the memory of a particular time-type can easily be found from the Time Manager, if the index is known. If only the name is known, this method provides a technique for obtaining the index from the name, thus allowing access to the memory address by name only.

With a name input, the function returns the index associated with that name. The default return value is -1, which is returned if no time-type by that name can be found.

In the event that a time-type with no defined name is used (the name "undefined" is the default name for all time-types), the return value is -2.

For each registered time-type, the *name* variable is compared with the lookup name. When a match is found, the index value is changed from -1 to the *index* variable of that time-type. The remainder of the time-types are still checked before returning. If a match is found and the index value is not equal to -1, there is a problem: the routine has found two occurrences of a time-type with the same name. It is not possible to distinguish between them, and the code terminates.

(k) **update**

This function is the master function once the simulation has begun. It is called from the S_define with the Simulator Time as its argument (i.e., *sys.exec.out.time*, or its equivalent for non-Trick simulations).

i. First, Simulator Time is updated, taking the value of the input argument.

ii. Next, the update functions are called on all subclasses of Time.

iii. The last step is necessary only in simulations in which time reverses direction; if time reverses at the current time, the offset between sim-time and TimeDyn will change.

Note that while steps *i. and ii.* are only followed if the argument is different from the previously recorded Simulator Time (*sim_time*), step *iii.* must be followed with every call. This is deliberate, and is essential due to the way time is updated during the dynamic integration procedures. These procedures can advance the *sim_time* value to the next Simulator Time before the simulation engine can process any commands at that Simulator Time; a command to reverse time will then be received after the time has been advanced to the instant at which the change should take place, causing a delay in its effect if step *iii.* were not processed.

(l) **verify_table_lookup_ends**

This method exists within all time representations, by inheritance from *Time* (on page 23). The Time Manager method runs through all registered time representations, calling the respective methods.

18. **Class: TimeManagerInit** **Inheritance:** Base-class

(a) **create_init_tree**

This function is used only during initialization; it creates the tree structure by which time-types are initialized to their starting values.

The user should specify a time-type to be used for initialization. If that is not done, or an invalid entry is made, then the following sequence of steps occur:

i. When the method *TimeManager::time_lookup* (on the preceding page) is called to find the initializer_index, none will be found and the initializer_index will return with value -1 (invalid entry) or -2 (not defined).

ii. An invalid entry will cause termination of the code in *verify_times_setup* (on page 39).

iii. An undefined entry will cause termination of the code in *verify_times_setup* (on page 39) if there is more than one time representation (i.e. anything in addition to Dynamic Time).

In the event that no initialization was defined, and only Dynamic Time is present, then Dynamic Time starts at 0.0, as it always does. No initialization is necessary. This method is processed trivially.

A "status" method is used to keep track of which time-types have been entered into the tree. Initially, all time-types are assigned a status value of 0. The Dynamic Time is then assigned value -2, and the initializer time the value 1, being the first level on the initialization tree (potentially overriding the Dynamic Time assignment).

Each time-type with status 0 is tested; if the user has carefully defined the tree with an "initialize_from" assignment for this time-type, the method *add_type_initialize*(on page 23) specific to that time-type is called to add the time-type to the tree. This function is iterative, it tests the parent type, and the parent's parent until it finds a type already in the tree (status > 0). If the chain terminates before a type already in the tree is found, no action is taken.

If the user has not carefully defined the tree, this function will look for the time-type with the lowest status >0 (i.e. highest on the tree) for which a valid converter exists to initialize the current time-type.

This search algorithm is processed as follows:

i. As long as one or more time-types remain to place in the tree, and all possibilities have not been exhausted, the following steps are taken.

   A. Set *num_added_pass* to 0.

   B. Increment *seeking_status*

      - *seeking_status* identifies the 'level' of the tree being tested.
      - On the first pass, (*seeking_status = 1*), time-types are tested for entry into the tree structure directly under the *initializer*.
      - On the second pass, (*seeking_status = 2*) the level beneath that is tested, etc.
      - If, at some level, nothing can be added (*num_added_pass = 0* at the end of the pass) then everything that can be added has been added; the algorithm is complete, or it has failed.

   C. For each time-type that is not yet in the tree:

      - If it has a defined time-type from which to initialize, add it to the tree (recursively adding its parent as necessary).
      - If it does not have a defined time-type from which to initialize, test it using status *seeking_status* to determine whether an appropriate converter function exists. If it does, add the type to the tree with status *seeking_status+1*. If not, wait for the next pass.

   D. The total number of types in the tree (*num_added_total*) is incremented by *num_added_pass*, and the code loops back to step A.

ii. If the code processes a path without further addition (*num_added_pass = 0*) and there is still something to add (*num_added_total* < *num_types*), then the algorithm has failed and the code terminates.

(b) **create_update_tree**

This function provides much the same functionality by the same method as that described in *create_init_tree* (on the preceding page), except that it considers the update tree. The only substantial difference comes in the recording of the tree; the initialization tree is recorded within TimeManagerInit, while the update tree is recorded in TimeManager. Also, while the initialization happens only once, the update happens repeatedly, so the update sequence is introduced and recorded. This necessitates an additional function, *set_ordered_update_list* (on this page).

(c) **organize_update_list**

While the user may specify times in any order in the S_define file, the Time Manager must have a record of the order in which to update the different representations. Since Dynamic Time updates directly from Simulator Time, it must be updated first. Then TAI is typically updated from Dynamic Time, and then the other derived times. While the initializer (*TimeManagerInit*) builds the tree for the update procedure, the Time Manager uses this function to reorganize the *time_vector*, which it then uses to ensure that representations are always updated before their respective dependents.

(d) **get_conv_ptr_index**

As each converter is registered, it is stored in a vector, with some index value. As each time representation is registered, it is also stored in a different vector, again with some index value. Each converter operates between two time representations. A variable *converter_ptrs_index* provides the lookup capability to find a particular converter given two time representations.

Consider the value $x = Ni_{\text{from}} + i_{\text{to}}$, with N representing the number of time-types, i_{from} the index of the representation from which the conversion is to be made, and i_{to} the index of the time representation to which the conversion is to be made. Then the x^{th} element in *converter_ptrs_index* is set to be an integer equal to the location of the converter in the converter registry that will convert from i_{from} to i_{to}.

This method receives the value x, and returns the index in the converter registry. An invalid input, or an input for which there is no registered converter causes the method to return a value -1.

Note that since converters are bidirectional the same result is obtained from $x = Ni_b + i_a$ and from $x = Ni_a + i_b$

(e) **get_conv_dir_init**

In addition to tracking the index of the converters, it is also necessary to track the versatility of those converters, and in which direction they should be used. For example, the converter between TAI and UTC (*TimeConverter_TAI_UTC*) can be used in either direction. Sending the value appropriate to a conversion from TAI to UTC into *get_conv_ptr_index* (on the current page) will return information on the index (and hence the memory location) of this converter, but not how to use it. There are two additional tables that provide directional information on usage of the converter, one for initialization (*init_converter_dir_table*), and one for updates (*update_converter_dir_table*).

While *converter_ptrs_index* is symmetric, the converter direction tables are not. Passing in $x = Ni_{TAI} + i_{UTC}$(from TAI to UTC) will return a value of +1, while passing in a value of $x = Ni_{UTC} + i_{TAI}$(from UTC to TAI) will return a value of -1. If a converter is not valid (e.g. Dynamic Time to TAI is valid at run-time, but not at initialization, while TDB to TAI is valid neither at runtime nor initialization), then the entry in the appropriate table has value 0.

This method returns the value for the converter at initialization.

(f) **get_conv_dir_upd**

See *get_conv_dir_init* (on this page) for details. This method returns the value for the converter at run-time.

(g) **get_status**

During the construction of the initialization and update trees (*create_init_tree* (on page 36) and *create_update_tree* (on the preceding page)), each time representation has a status corresponding to its location in the tree. This function returns the status of any given time representation.

(h) **increment_status**

During the construction of the initialization and update trees (*create_init_tree* (on page 36) and *create_update_tree* (on the preceding page)), each time representation has a status corresponding to its location in the tree. This function sets the status of a time-type to be one higher than that of its parent.

(i) **initialize**

Sets the references *dyn_time_index* and *initializer_index.*

Calls *verify_times_setup* (on the current page) to ensure that the time-type settings are mutually consistent.

Allocates memory for several arrays, and initializes some values.

(j) **initialize_manager**

Comprises a series of calls, all to methods within this class:

i. *initialize*
ii. *populate_converter_registry*
iii. *verify_converter_setup*
iv. *create_init_tree*
v. *initialize_time_types*
vi. *create_update_tree*

(k) **initialize_time_types**

This method starts by initializing the head of the initialization tree with the appropriate call to *initialize_initializer_time.* Then, it progresses through the remaining time representations calling their respective *initialize_from_parent* methods on any time types that have not been initialized. This method is iterative, and calls itself on the parent (in the initialization tree) if the parent has not been initialized. Eventually, it must find an initialized representation (since the head of the tree has been initialized).

(l) **populate_converter_registry**

This method sets the values of the converter tables discussed in *get_conv_ptr_index* (on the facing page), *get_conv_dir_init* (on the preceding page), and *get_conv_dir_upd* (on the facing page).

(m) **set_status**

During the construction of the initialization and update trees (*create_init_tree* (on page 36) and *create_update_tree* (on page 37)), each time representation has a status corresponding to its location in the tree. This function sets the status of a time-type to some input value.

(n) **verify_times_setup**

Makes a number of sanity checks that the time-types are mutually compatible and that there are no inconsistencies in the way the time module was set up. In particular, it ensures that the simulation can be initialized, and that there are no duplicate time representations.

(o) **verify_converter_setup**

This method verifies that those converters that need special considerations (at this time, this only applies to those that use data lookup tables) are appropriately configured.

19. **Class: TimeMessages** **Inheritance:** This is the message handler for the Time Representations Model. It contains no methods.

20. **Class: TimeMET** **Inheritance:** *TimeUDE* (on page 47)

Mission Elapsed Times have the built-in capability to 'hold' their progression for some period. Beyond the typical UDE data, this class contains an additional boolean value, *hold.*

(a) **update**

MET contains two values that indicate whether a hold is active, *hold* and *previous_hold*. The former is set externally, the latter is set within this method and provides a comparison data point to identify when *hold* has been changed.

If MET has just transitioned to a hold, the regular update method is run to bring MET up-to-date with the time at which the hold was commanded, and *previous_hold* is transitioned to *false*.

If MET is in a hold, and was previously in a hold, no action is taken.

If MET is not in a hold, and was not previously in a hold, the inherited update method is called.

Finally, if MET has just transitioned from a hold, the time converter is updated to reset the offset between this type and its parent. The actual value of MET does not require updating, because it retains its current value until the next time-step.

Note that when the hold flag is being toggled, it is beneficial to set the Time Manager's *simtime* value to 0. The Time Manager compares that value to the Simulator Time to determine whether to call the individual update methods; setting it to zero forces the update method. Without this step, this update method may be bypassed, and processed at the time-step after it was intended.

21. **Class: TimeStandard** **Inheritance:** *JeodBaseTime* (on page 23)

(a) **add_type_initialize**

This function is intended to add a time-type to the initialization tree if its parent is already in the tree, and to recurse on its parent if the parent is not in the tree.

Initially, two values are set:

i. The status of the time-type is changed from 0 to -1 for reasons explained later.

ii. Then, the *time_types_lookup* (on page 35) function is called to obtain the index value of the parent type.

This function causes the code to terminate on any of the following three conditions:

i. The parent type cannot be found. This function is only called if the user has named the parent type. If the user has done so, but not registered the type with the Time Manager, there is the potential for serious malfunction.

ii. The status of the parent is -1. This can only be achieved when the parent has already been processed by this function within the current recursion chain, which indicates that the user has specified a circular initialization path.

iii. If no converter exists to convert the parent time into the child time-type. The user has specified that this converter should be used, but failed to provide the converter.

The next step is the recursion component: if the parent is not in the tree (status = 0), and has a defined parent itself, then carry out this same function on the parent.

If the parent is in the tree (either by being in the tree originally, or by a result of the previous command), then add the type and increment *num_added_pass*. If not, set the status of the type back to 0 (indicating that the type has not been added to the tree), and return to the calling function.

Note – if this was called recursively, the return of status 0 will cause the child's status to go to 0, all the way down the chain until it returns to create_init_tree.

(b) **calculate_calendar_values**

Converts the time in any representation from a Truncated Julian format into a Gregorian calendar format. The algorithm for this conversion is non-trivial.

The calendar format has both regular and irregular cyclical variations: there are always 60 seconds in a minute, 60 minutes in an hour, and 12 months in a year. There are 24 hours in a day, except for the addition of leap seconds, although these can be factored in relatively easily. There are 28-31 days per month, with known predictability. There are 365-366 days in a year, with leap years occurring in a known pattern: there is a leap year every 4 years, excepting one every 100 years, excepting one every 400 years (there are 24-25 leap years every 100 years, and always 97 leap years in a 400-year period). There are 146,097 days in any 400-year period.

The most difficult challenge in converting from Julian to Gregorian is in identifying the month number. This can be achieved through integer division (which is faster than a lookup) if care is taken to avoid the irregularities associated with the leap-day February 29, and the uneven nature of the days-per-month distribution.

[A basic application of integer division fails almost immediately: with 31 days in January, the divisor must be > 31 and < 32 to produce the same answer for days 1-31 but not for day 32. Then, the same answer is produced for days 32-62, but day 62 is either March 2 or March 3, not February as intended.]

The first of these irregularities can be accounted for if the starting time is chosen carefully. To avoid an "if-else" clause on the leap-day, it makes sense to make the leap-day the last day of the year. Then, if the year requires 366 days, it is added automatically, and if the year has only 365 days, there is no February 29 because the date rolls to the next year first. Consequently, the epoch is placed on March 1. Furthermore, it should be on March 1 immediately after a leap-day so that some residual partial day can build until a leap-day is needed after 4 years. The 100-year and 400-year cycles complicate this, and by the same argument, the epoch should be on March 1 after a 400-year leap-day. It is chosen then that the epoch date be March 1, 1600.

The epoch date for Truncated Julian Time (May 24, 1968) happens to occur 134,493 days after the March 1, 1600, epoch.

After obtaining the year information from the Julian Date, the second irregularity (uneven nature of days-per-month) is accounted for by adjusting the epoch time slightly. This is explained in step 10 ("x.") below.

The algorithm for converting Truncated Julian Time into Gregorian calendar then is described by:

i. Take the integer part of the Truncated Julian Time $TJT_{int} = int(TJT)$, where TJT represents Truncated Julian Time, and TJT_{int} the integer part thereof.

ii. The fraction of the current day that has elapsed is first represented as a number of minutes. $min = 1440\,(TJT - TJT_{int})$

iii. The fractional part of the last minute provides the seconds value of the clock. $seconds = 60\,(min - int(min))$

iv. An additional term, *clock_resolution* is used to force *seconds* to 'tick-over' to 0 if the value is sufficiently close to 60. This keeps the data-stream more uniform. If the clock is forced to 'tick-over', that requires that the minutes value be incremented by 1, and a check made as to whether that action also caused the day to 'tick-over'.

(*e.g. July 7 23:59:59.999999999999 = July 8 0:0:0.0*)

v. The hour of day is found from an integer division, by 60, of the integer representation of minutes-of-day. Then the clock minutes value is equal to the remainder.

$$hour = \frac{min_{\text{int}}}{60},$$
$$minute = min_{\text{int}} - 60 \cdot hour.$$

vi. If the value *julian_day* (the integer part of TJT + any increment from step iv) has not changed from its previous value, the algorithm is completed. If it has, the more cumbersome task of determining the date is undertaken.

vii. Calculate *julian_day* by adding 134,493 to adjust for epoch,
$julian\_day = TJT_{\text{int}} + 134493$.
(134,493 represents the number of days from the conversion epoch to the Truncated Julian epoch).
March 1, 1600, is day 0, not day 1; this is not an arbitrary decision, but necessitated by the mathematical structure.

viii. Calculate *n_400,* the number of 400-year periods since March 1, 1600, by dividing the number of days by 146,097 (days per 400-year period). Since March 1, 1600, is day 0, then March 1, 2000, is day 146,097, and falls into the next 400-year period.

ix. Calculate *r_400,* the number of days since the end of the last 400-year period (i.e., since March 1, 1600, or March 1, 2000). In the next step, it becomes necessary to treat March 1 as day 1 rather than day 0; hence we also add 1:

$$r_{400} = day - 146097 n_{400} + 1 \quad , \quad r_{400} \in [1, 146097]$$

x. Calculate *n_100,* the number of 100-year periods in the current 400-year period. The first three 100-year periods have 36,524 days, the fourth has 36,525 days.
In the case that March 1 was represented by day 0, the last day in the fourth period would be day 146,096. A suitable divisor would have to be $x > 36524$ to make $(146096/x) < 4$. But then day 36,524 would fall into the first period, when it should be the first day in the second period.
In the case that March 1 is counted as day 1, then the last day in the fourth period is 146,097. Once again, $x > 36524$, but now day 36,524 correctly falls into the first period, as the last day. $x = 36524.3$ is an appropriate number such that the last days in each period are 36,524; 73,048; 109,572; and 146,097. Because this divisor is non-integer, integer arithmetic cannot be utilized; instead, cast the result to an integer, in effect taking the integer part of the result.

$$n_{100} = int(r_{400}/36524.3)$$

xi. Calculate *r_100*, the number of days since the last 100-year period (March 1, 1900, or March 1, 2000). There are 36,524 days in each of the first three 100-year periods in any 400-year period, so subtract the appropriate multiple of 36,524. In the next step, the last period is no larger than earlier periods, and integer arithmetic can be utilized if March 1 is counted as day 0. Hence, the extra day is removed again.

$$r_{100} = r_{400} - 36524 n_{100} - 1 \quad , \quad r_{100} \in [0, 36524].$$

xii. Calculate *n_4*, the number of 4-year periods. There are 1,461 days in the first 24 of these, and 1,460 days in the 25$^{\text{th}}$, unless $n_{100} = 3$ in which case the 25$^{\text{th}}$ period also has 1,461 days.

$$n_4 = r_{100}/1461 \ , \ \ n_4 \in [0, 24]$$

xiii. Calculate

$$r_4 = r_{100} - 1461 n_4 + 1 \ , \ \ r_4 \in [1, 1461].$$

The next step has the last period being larger than the earlier periods again, so again add 1 to make March 1 be represented as day 1.

xiv. Calculate the number of whole years in the current 4-year period. Again, this real result is cast to an integer.

$$n_1 = r_4/365.3$$

xv. Calculate

$$r'_4 = r_4 + 2n_1 + 30,$$

a value to be used in evaluating the month number. Because the month length fluctuates, the epoch is adjusted such that March 1, 2000, is day 31. Two additional days are added each year, so that February 28, 2001, is day 395, and March 1, 2001, is day 398. This unusual strategy effectively gives February 30 days, smoothing out the month-to-month fluctuations. It also puts the epoch on February 1, 1600.

xvi. Calculate

$$m' = \text{int}\left(\frac{r'}{30.585}\right)$$

With approximately 30.585 days per month, the number of months in the 4-year period can be determined.

xvii. Calculate the month number.

$$m = m' + 2 - 12\left(\frac{m'+1}{12}\right)$$

This first adjusts *m'* accounting for February *(m=2)* being counted *as m' = 0, 12, 24, 36, 48.* The second adjustment accounts for the whole years; the integer division in the parentheses "rolls" each January *(m' = 11, 23,35,47),* reducing the value of m from its December value.

xviii. Calculate *d*, the day number on the current month. With 30.585 days per month,

$$d = r' - \text{int}\left(30.585 m'\right)$$

with the second term being an integer equal to the cumulative number of days at month's end (February – 30, March – 61, April – 91, May – 122, etc.)

xix. Calculate *y*, the year.

$$y = 1600 + 400 n_{400} + 100 n_{100} + 4 n_4 + \left(\frac{m+1}{12}\right)$$

The calendar could now be expressed as yyyy/mm/dd::hr:min:sec.sec

This is usually only necessary when outputting values in a required format; the simulation runs on Truncated Julian Time only, with no reference to the calendar time. Consequently, this function is not called as part of the regular time update.

(c) **calendar_update**

Each Standard Time has several potential ways of being presented, including purely decimal values (seconds since epoch, days since epoch, Truncated Julian Time), and a calendar. The decimal representations are more computationally efficient, and are maintained in current status throughout the simulation. Maintenance of the calendar representation is more time-consuming, and is only called as needed on a type-by-type basis. This function checks whether the simulation has advanced since the last calendar update. If it has, *calculate_calendar_values* (on page 41) is called to convert the decimal representation into a calendar representation.

(d) **convert_from_calendar**

This function is intended for use during initialization, when data values are read in as a calendar format (more commonly used), and must be converted to a a decimal format for propagation within the simulation.

Six values are input – year, month, day, hour, minute, and second.

The algorithm basically follows the inverse process of that presented in *calculate_calendar_values* (on page 41).

i. The number of years since epoch is

$$y = year - 1601 + \left(\frac{month + 9}{12}\right)$$

(notice that this integer division term "rolls" each March)

ii. The value, y, can then be divided into a number of 400-year periods, plus a number of 100-year periods, etc.

$$\begin{aligned} n_{400} &= \frac{y}{400} \\ n_{100} &= \frac{y - 400n_{400}}{100} \\ n_4 &= \frac{y - 400n_{400} - 100n_{100}}{4} \\ n_1 &= y - 400n_{400} - 100n_{100} - 4n_4 \\ m &= month - 2 + 12\left(1 - \frac{month + 9}{12} + n_1\right) \end{aligned}$$

m is the month number in the 4-year period starting March 1.

iii. The Truncated Julian Time can now be calculated:

A. The fractional part of the day is

$$TJT_1 = \frac{second}{86400.0} + \frac{minute}{1440.0} + \frac{hour}{24.0}$$

B. Each day counts as "1"

$$TJT_2 = TJT_1 + day$$

C. The month value, m, is complicated. In *calculate_calendar_values* (on page 41) the method was developed by which the epoch was adjusted by 30 days (actually, by 31, because 1 day had already been added to make March 1 day 1 instead of day 0), and 2 days added each year to smooth out the month-to-month variations. With these modifications, it was possible to carry out integer arithmetic, using 30.585 days per month and taking the integer part of the result. That same method can be used here: the number of days is calculated from m, then the 2 days per year and the 31 day-offset removed.

$$TJT_3 = TJT_2 + (int\,(30.585m) - 2n_1 - 31)$$

D. By setting the epoch to March 1, 1600, the number of days in each completed 4-year, 100-year, and 400-year period is known and fixed. The final step is to remove the 134493 day offset between March 1, 1600, and May 24, 1968, the epoch of Truncated Julian Time.

$$TJT = TJT_3 + 1461n_4 + 36524n_{100} + 146097n_{400} - 134493$$

Note – n_1 is not included because m counts the number of months in a 4-year period.

iv. Finally, days since epoch is simply the difference between current Truncated Julian Time, and the stored value of Truncated Julian Time at the epoch. Seconds since epoch is then trivially calculated.

(e) **initialize_from_parent**

Takes the value of the time in the representation that is parent to this one in the initialization tree, and converts it to this time. This is a recursive function, and will call itself on the parent type if the parent has not yet been initialized.

(f) **initialize_initializer_time**

Typically, the simulation starts at some prescribed time in some prescribed time representation. This function takes the input data and uses it to initialize the prescribed time representation (in the event that it is a Time Standard). There are several formats in which that initial time can be presented:

i. **calendar**
Calls *convert_from_calendar* (on the preceding page) to generate the decimal representation.

ii. **Julian**
Converts directly to Truncated Julian, and from there to days and seconds.

iii. **Modified Julian**
Converts directly to Truncated Julian, and from there to days and seconds.

iv. **Truncated Julian**
Converts to days and seconds.

v. **Seconds since epoch**
Converts to days and Truncated Julian Time.

vi. **Days since epoch**
Converts to seconds and Truncated Julian Time.

If the user does not specify the format, the code will attempt to interpret what was intended, but any ambiguity will result in termination.

(g) **julian_date_at_epoch**
Simply returns the epoch as a Julian, rather than Truncated Julian value.

(h) **seconds_of_year**
This method relies on the calendar functionality included with JEOD v5.1. Two variables are included in the Time Standard class, *seconds_at_year_start*, and *year_of_last_soy*. The former records the value of the variable *seconds* at the beginning of the current year; the latter records to which year *seconds_at_year_start* pertains.
First, the calendar is brought up-to-date (if it is not already so). Then the code divides into 3 paths:

i. When *calendar_year* = *year_of_last_soy* (most frequent path). This path simply differences the current value of *seconds* from the value at the start of the year.
ii. When *calendar_year* = *year_of_last_soy* + *1* AND *calendar_month* $\leqslant 2$ (unusual). This path requires that the year has ticked over once since the last time this method was accessed, and the current month is January or February. The value *seconds_of_year* is calculated directly, and used to define a new value for *seconds_at_year_start*; *year_of_last_soy* is also redefined.
iii. All other cases. This path is most often accessed during the first call to this method for any of the time-types. Otherwise, it would be a rare circumstance that would lead to this path. It temporarily reassigns the current time to be the start of the year, calculates the *seconds* value there, then returns to current time. The calculated value becomes *seconds_at_year_start*, the current year becomes *year_of_last_soy*, and the calculation of *seconds_of_year* is then trivial.

(i) **set_time_by_seconds**
Uses the version in *JeodBaseTime* (on page 23), and adds conversion to Truncated Julian Time.

(j) **set_time_by_days**
Uses the version in *JeodBaseTime* (on page 23), and adds conversion to Truncated Julian Time.

(k) **set_time_by_trunc_julian**
Given a Truncated Julian Time, converts it into days since epoch and seconds since epoch.

22. **Class: TimeTAI** **Inheritance:** *TimeStandard* (on page 40).
Inherits completely from *TimeStandard*.

23. **Class: TimeTDB** **Inheritance:** *TimeStandard* (on page 40)
Inherits completely from *TimeStandard*.

24. **Class: TimeTT** **Inheritance:** *TimeStandard* (on page 40)
Inherits completely from *TimeStandard*.

25. **Class: TimeUDE** **Inheritance:** *JeodBaseTime* (on page 23).

(a) **add_type_initialize**

Works in much the same way as the *TimeStandard* (on page 40) version of the method of the same name, but adds additional protection to verify that the representation that defines the epoch (as well as the parent) of this representation is configured correctly and already established in the initialization tree, and that the representation from which this will be updated at run-time is defined and available.

(b) **clock_update**

Given the decimal representation of time elapsed since epoch, this method generates a clock (day::hour:minute:second) representation.

(c) **convert_epoch_to_update**

In the case where a UDE has an epoch defined in one representation, and ticks with (i.e. is updated from) another (e.g. a clock may start at some known UTC time, but tick with TAI), this method temporarily converts the specified epoch time into the "ticks-with" representation, in order to calculate the offset between this representation and the "ticks-with" representation.

That offset is then used by the converter to provide the standard conversion at run-time.

The only complicating issue in this routine occurs when the converter between the epoch-definition type and the "ticks-with" type has already been initialized. Under some circumstances, the initialization of a converter establishes the converter for a restricted period of time, and the epoch could lie outside that range. To be safe, the converter is de-initialized, and re-initialized at the epoch value, then de-initialized again (if it is needed later, at run-time, it will be re-initialized with values appropriate for the simulation time).

(d) **initialize_from_parent**

This function is used in the situation in which a simulation is initialized with respect to some other known time-type, and this time-type representation fits somewhere – other than at the head – in the initialization tree.

The *parent* in this case is the representation from which this one will be updated at run-time. If that has not been initialized, this function (or its parallel in *TimeStandard* (on page 40), as appropriate) is called for the *parent.*

There are two methods by which a UDE can be initialized when it is not at the head of the initialization tree – either by specification of its value at simulation-start, or by specification of an epoch.

In order to initialize a UDE by specifying the epoch, it is necessary to temporarily step out of the current time, and go to the epoch time of the UDE. First, the current value of the *parent* is stored off, then the algorithm branches in a multi-path solution.

i. The simplest case is one in which the *parent* is also the representation in which the epoch is defined; in that case the *parent* value at simulation-start is overwritten with the *parent* value at the epoch.

ii. In the situation that the two differ, the following procedure is used:

A. A converter is found between the epoch-defining type and the *parent* type, and initialized if necessary. If no converter exists, the code must terminate. Note:

this step is only carried out here in the case that the epoch-defining type is itself a UDE; otherwise, this step is more complicated and is carried out in *convert_epoch_to_update* (on the preceding page), but it must be carried out at some point.

B. If the epoch-defining type has already been initialized, its value must also be stored off temporarily and overwritten with the value at epoch.

C. The epoch-defining type is then populated with the epoch definition, and converted to the *parent* type using *convert_epoch_to_update* (on the previous page). At this time, the *parent* defines the epoch just as well as the epoch-defining type does; the epoch-defining type can now be reverted to its former configuration, and we can proceed as though the epoch-defining type and the parent type were the same.

The final step requires the initialization of the converter between the UDE and the *parent.* For this, we can use the current values of both:

i. The UDE has value 0.0 if the initialization is based on an epoch definition, and some other defined value if the initialization is based on an initial value.

ii. The *parent* has value equal to that at the UDE's epoch if the initialization is based on an epoch definition, and the appropriate value at simulation start if the initialization is based on some initial value.

Either way, the offset between the two types can now be calculated, and used throughout the rest of the simulation.

If the *parent* configuration has been altered during the course of this routine, it must now be reverted to its original configuration, and that value used to initialize the UDE for the time corresponding to the start of the simulation.

(e) **initialize_initializer_time**

This method is used to initialize the UDE time in the case that the UDE time is being used to initialize the simulation (i.e., it is at the head of the initialization tree). To properly initialize the UDE in this situation, it must have an epoch defined in some other representation, and an initial value at simulation-start.

One additional requirement on this type of initialization is that the epoch cannot be defined as being in some other UDE, and if there are any Standard Times in the simulation the epoch must be a Standard Time (the epoch can be defined in Dynamic Time, but only if there are no Standard Times included).

Verification is made that the two variables *TimeManagerInit::sim_start_format* and *TimeUDE::initial_value_format* are consistent, and the initial values are then set as defined.

Verification is then performed on the epoch definition, ensuring that the epoch-defining type and epoch values are defined appropriately, and that there is no conflicting definition in the value of the epoch.

Next, the initializing value of the UDE is converted into the *parent* representation (the clock from which the UDE will be updated at run-time). To initialize the converter for this purpose, it is usually necessary to first consider the epoch values, since data must be known at both ends of the converter before it can be initialized. For this representation, the initial values are stored, then it is set to zero. The epoch-defining representation is set

to its predefined value. If the *parent* and epoch-defining representation are different, the epoch-defining values are converted to *parent* values, thus providing the value for parent and this representation at the same instant in time. Once the converter is initialized, the initial values for this representation are restored, and used to set the simulation-start values in *parent*.

(f) **must_be_singleton**

Returns *false*, there may be multiple instances of UDE times.

(g) **set_epoch_dyn**

Makes several checks that the epoch format and epoch values are consistent with an epoch defined in Dynamic Time, then sets the value of Dynamic Time to that specified as the epoch.

(h) **set_epoch_std**

Makes several checks that the epoch format and epoch values are consistent with an epoch defined in a Standard Time, then sets the value of that time to that specified as the epoch.

(i) **set_epoch_times**

Calls one of

- set_epoch_dyn
- set_epoch_std
- set_epoch_ude

depending on whether the epoch is defined in Dynamic Time, in a Standard Time, or in another UDE time.

(j) **set_epoch_ude**

Makes several checks that the epoch format and epoch values are consistent with an epoch defined in a UDE Time, then sets the value of that time to that specified as the epoch.

(k) **set_initial_times**

Verifies that the initial value of this representation is not over-constrained (e.g. by setting both seconds-since-epoch and clock-seconds), then propagates the specified value to other formats (e.g. converts a clock format to days-since-epoch and seconds-since-epoch).

(l) **set_time_by_clock**

Converts a clock input format into days-since-epoch and seconds-since-epoch.

(m) **set_time_by_days**

Converts a days-since-epoch input format into and seconds-since-epoch and a clock.

(n) **set_time_by_seconds**

Converts a seconds-since-epoch input format into and days-since-epoch and a clock.

(o) **verify_epoch**

Verifies that the epoch representation, the values specified for the epoch, and the format in which they are specified are appropriate.

(p) **verify_init**
Verifies that the initial values specified for this representation, and the format in which they are specified are appropriate, then calls *set_initial_times* to assign those values.

(q) **verify_update**
Verifies that the *parent* representation is defined appropriately.

26. **Class: TimeUT1** **Inheritance:** *TimeStandard* (on page 40).
Inherits almost completely from *TimeStandard*, with only one additional method:

(a) **get_days**
Simply returns the value *days* (days-since-epoch).

27. **Class: TimeUTC** **Inheritance:** *TimeStandard* (on page 40).
Inherits completely from *TimeStandard*

3.3.3 Default data files

This section describes the data files included with the release of JEOD v5.1.

1. **tai_to_ut1.cc**

   This file gives a tabulation of the differences between *TAI* and *UT1* as a function of *TAI*.

   Instructions for updating this file are contained in the Extension section of the User Guide (chapter 4).

2. **tai_to_utc.cc**

   This file gives a tabulation of the differences between *TAI* and *UTC* as a function of *UTC*.

   Instructions for updating this file are contained in the Extension section of the User Guide (chapter 4).

3.3.4 Extensibility

The Time Representations Model is designed in such a way that it can easily be extended with the addition of more time representations.

Each new time representation must meet the following requirements:

1. It shall be included as a class of its own, inheriting from JeodBaseTime or TimeStandard.
2. It shall have a name
3. It shall have a constructor and destructor
4. There shall be some method for deriving its value from one of the time classes that already exist.

A new time representation must have a new Time Converter. A user may also wish to define a new Time Converter to convert directly between two preexisting time representations (e.g., UT1 to MET).

Each new Time Converter must meet the following requirements:

1. A converter shall be defined to allow the derivation of the new time representations.
2. The converter shall be included in a class of its own, inheriting from *TimeConverter*.
3. The converter class shall define one or both of the methods
   (a) *convert_a_to_b*.
   (b) *convert_b_to_a*.
4. The converter class shall define the *valid_directions* attribute to specify when *convert_a_to_b* and/or *convert_b_to_a* is valid.
5. The converter class shall have an *initialize* method.
6. The converter class shall have a constructor and a destructor.

Instructions for adding these methods are found in the Extension section 4.3 on page 74 of the User Guide.

3.4 Inventory

All Time Representations Model files are located in the directory
`${JEOD_HOME}/models/environment/time`. Relative to this directory,

- Header and source files are located in the model `include` and `src` subdirectories. Table **??** lists the configuration-managed files in these directories.
- Data files are located in the model `data` subdirectory. See table **??** for a listing of the configuration-managed files in this directory.
- Documentation files are located in the model `docs` subdirectory. See table **??** for a listing of the configuration-managed files in this directory.
- Verification files are located in the model `verif` subdirectory. See table **??** for a listing of the configuration-managed files in this directory.

Chapter 4

User Guide

The User Guide is divided into 3 components, one for each of three different types of user.

The Analysis section of the user guide is intended primarily for users of preexisting simulations. It comprises the following elements:

- A description of how to modify Time Representations Model variables after the simulation has compiled, including an in-depth discussion of the input file;
- An overview of how to interpret (but not edit) the S_define file;
- A sample of some of the typical variables that may be logged.

The Integration section is intended for simulation developers. It describes the necessary configuration of the Time Representations Model within an S_define file, and the creation of standard run directories. The latter component assumes a thorough understanding of the preceding Analysis section of the user guide. Where applicable, the user may be directed to selected portions of Product Specification (Chapter 3), or to examples in the tutorial or Verification and Validation chapter (Chapter 5).

The Extension section is intended primarily for developers needing to extend the capability of the Time Representations Model. Such users should have a thorough understanding of how the model is used in the preceding Integration section, and of the model specification (described in Chapter 3).

4.1 Analysis

4.1.1 Overview of the Time Representations

Typically, a simulation will contain a *time* object, which will contain a time manager, and possibly some additional number of time representations. There are three fundamentally distinct time concepts working in JEOD:

- Simulator Time

- Dynamic Time
- Derived Time

Simulator Time provides the behind-the-scenes simulator control. For Trick users, this is the value *sys.exec.out.time.* It provides the information necessary to ensure that functions are called in the appropriate order, that tasks are queued appropriately, and to query whether a task has already been completed. In general, it must be incremental, i.e. always advancing forwards. Simulator Time is not a part of the Time Representations Model, since it does not measure a physical passage of time.

Dynamic Time is the time used for representing the physics. When physical constants rely on a timescale (e.g. speed of light, meters *per second*), it is the Dynamic Time to which they refer. Consequently, this is the time used by the integrators. The Dynamic Time is automatically included with the Time Manager.

Derived Time has unbounded possibilities, with the ability to represent any clock that the user chooses, as long as that clock is somehow related to Dynamic Time. To find which times are available, open the S_define file, find the *time* object, and look for statements similar to these following:

```
environment/time:  TimeTAI   tai;
environment/time:  TimeUTC   utc;
environment/time:  TimeUT1   ut1;
```

There will be one of these statements for each additional Derived Time. Within the concept of Derived Time, there are two sub-concepts:

- Standard Time
- User-Defined Time

These are described in more detail below.

4.1.1.a Dynamic Time (*manager.dyn_time*, class *TimeDyn*)

The Dynamic Time must be present in any simulation, but will not appear by itself in the S_define file. It always has initial value 0.0, and counts the number of SI seconds (*manager.dyn_time.seconds)* elapsed since the simulation began. Note that this may not be the same as the Simulator Time (e.g. *sys.exec.out.time*), since Dynamic Time has the capacity to run at different rates – even in reverse – whereas the simulator clock is simply an incremental counter. The simulation dynamics are all based on Dynamic Time.

4.1.1.b Standard Times (TimeSTD)

The concept of a Standard Time is that it is commonly accepted, with no ambiguity. Picking a particular value on a particular clock (e.g. noon on January 23, 2009, UTC) is well understood

without additional context. Consequently, there can be only one clock running in a simulation for each of these times (any more would be redundant). JEOD 2.0 is released with the following commonly used Standard Times, although this list may have been extended by your simulation developer.

- TAI (International Atomic Time) is the most closely linked to Dynamic Time. It ticks at the same rate, but starts with a simulation-specific initial value, defined by the user. Most derived times are derived from TAI; this is found in almost all simulations.

- UTC (Coordinated Universal Time) is the standard clock on Earth. If initializing the simulation at a particular time, this is usually handled with UTC. It ticks at the same rate as TAI, but occasionally has leap seconds introduced to keep it near-synchronous with UT1. The historical record of the occurrence of leap seconds is provided. If a simulation happens to span a time at which a leap second was introduced, UTC will be updated accordingly. The user may override the value corresponding to the offset between TAI and UTC, as described in the *data override section* (on page 73).

- UT1 (Universal Time) is based on the solar day (whereas UTC is based on days of 86,400 SI seconds). It ticks at an irregular rate, a result of a variable rotation rate of Earth, caused by tidal friction, polar motion, and tectonic action, among others. The difference between UTC and UT1 is therefore variable, and not highly predictable, although it is always less than 1 second (when the differences become too large, a leap second is added to UTC to bring them close again – see figure 4.1. For historical simulations, calibrated data is provided to convert directly from TAI to UT1. For simulations outside the range of the provided data, the offset will be held at the last known value; no attempt is made to predict offsets. Instructions for overriding this value are provided in the *data override section* (on page 73). Instructions for updating the calibrated data set with the most recently available data are provided in the *data update section* (on page 72).

  Special Notes Re: previous versions of JEOD:

  1. In this version, the offsets between TAI and UTC, and between TAI and UT1, are updated continuously, whereas the time management in JEOD 1.x.y identified the offset at the start of the simulation only, and used that offset throughout the simulation. For backward compatibility, the new updates can be turned off by setting the flags true_utc and true_ut1 to false. This should only be done for tests that require backward compatibility for comparison to simulations conducted in JEOD 1.x.
  2. In JEOD 1.x.y, the data for generating UT1 was simply the DUT1 value - the difference between UTC and UT1. In JEOD 2.0, UTC is no longer the "fundamental" second, and the preferred conversion method to get UT1 is to do so from TAI. Therefore, simply using DUT1 to override the tai_to_ut1 conversion value will produce error. The conversion must be from TAI to UT1, therefore attempting to override the data requires additional consideration of the number of leap seconds (UTC to TAI conversion) in order to generate the TAI to UT1 conversion. Alternatively, a converter could easily be written to generate UT1 from UTC directly, but this has been deliberately omitted from

the general release to avoid confusion over identifying the preferred method for generating UT1. Overriding the data is not recommended for general practice.

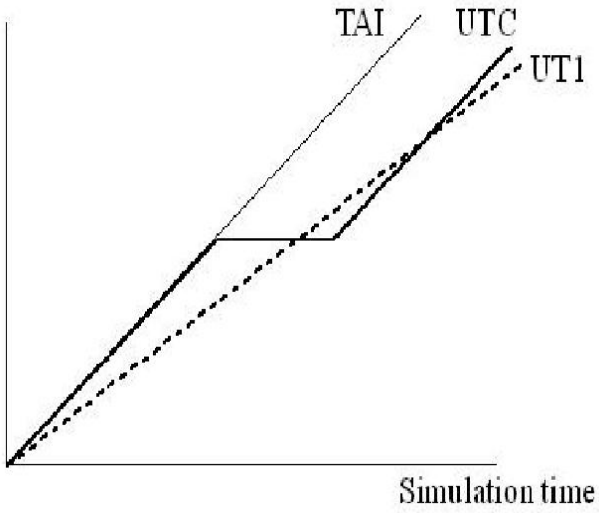

Figure 4.1: Illustration of the relative evolution of TAI, UTC, and UT1 values, and how leap seconds are used.

- GMST (Greenwich Mean Sidereal Time) is based on a sidereal clock (1 sidereal day = 1 rotation period, approximately 23 hours and 56 minutes on a synodic clock). There are 24 sidereal hours in a sidereal day, 60 sidereal minutes in a sidereal hour, and 60 sidereal seconds in a sidereal minute, although these are rarely – if ever – used.
- GPS (Global Positioning System time) ticks at the same rate as TAI and is simply offset by a known value, but counts in weeks and seconds of week rather than days, or the conventional calendar found in TAI.
- TT (Terrestrial Time) ticks at the same rate as TAI, and is simply offset by a known value. It is used in ephemeris models.
- TDB (Barycentric Dynamic Time) ticks at an obscure rate that accounts for the relativistic corrections introduced by Earth's orbital eccentricity. Since TAI is geocentric (the clocks are located at the bottom of Earth's gravitational well, and move with Earth), it is not the best clock for solar-system missions; TDB removes the geo-effects from TAI.

4.1.1.c User-Defined Times (TimeUDE)

The main distinction between a Standard Time and a User-Defined Time is that User-Defined Times are ambiguous without additional context. A commonly used example of a UDE (User Defined Epoch) is Mission Elapsed Time. Simply stating that a simulation started two hours after the mission started is not a very meaningful statement, unless we also have information on when the mission started. The *epoch* of UDEs (i.e. when their value was zero) must be defined before

they make any sense at all; that epoch must ultimately be anchored with a Standard Time, or with the Dynamic Time. A secondary difference arises from the ambiguity of a UDE time; while Standard Times are restricted to one instance per time-type, the UDEs have no such restriction. A dozen or more different clocks could be running simultaneously, representing different time zones (ticking with UTC), or different mission-elapsed times for simulations involving multiple vehicles.

There are two types of UDE:

- UDE: These have an epoch defined in one time type, and tick in lockstep with another (may be different).
- MET (Mission Elapsed Time): This special type of UDE also has the ability to halt periodically, to allow for occurrences such as pre-launch hold points. The basic UDE will keep ticking through such hold points.

4.1.2 Available Data

This section describes the output data that a user is most likely to need, identified by the major classes of data.

4.1.2.a JeodBaseTime

JeodBaseTime is the base class of all time representations. It contains the following variables (among others):

- *name* simply names the time object
- *days* represents the current time as some decimal number of elapsed days since epoch
- *seconds* represents the current time as some decimal number of elapsed seconds since epoch
- *initial_value* is the value of *seconds* at the start of the simulation.

4.1.2.b Dynamic Time (time_dyn)

Dynamic time inherits those data from JeodBaseTime and simply counts SI seconds, and days, elapsed since the start of the simulation. It also contains a variable, *scale_factor*, that allows it to deviate from the simulator time (*e.g. sys.exec.out.time*), although changing this value is not recommended for a beginning user.

4.1.2.c Standard Times

The Standard Times also inherit from Time, and add the following variables:

- *calendar_year*

- *calendar_month*
- *calendar_day*
- *calendar_hour*
- *calendar_minute*
- *calendar_second*
- *trunc_julian_time*
- *julian_date*
- *tjt_at_epoch*

Truncated Julian Time (*trunc_julian_time*) is used as the basic anchor for each clock, giving each one a basis for measuring "absolute" time, and setting the offsets between the clocks. It represents the number of days elapsed since midnight on May 23 / 24, 1968, as measured in its own clock. That means that Truncated Julian Time = 0 in Terrestrial Time (TT) does NOT represent the same instant as Truncated Julian Time = 0 in Universal Coordinated Time (UTC).

Julian Date is the original basis for Truncated Julian Time, but has its epoch much farther in the past, so has fewer available signigificant digits for precision work. Like Truncated Julian Time, it has a distinct value in each clock at any specific instant in time, and is related to Truncated Julian by:

$$Julian = Truncated - Julian + 2440000.5$$

in all clocks. Because of its loss of precision, Julian Date should be used for reference purposes only.

In most Standard Times (excepting GPS and GMST), *seconds* and *days* are the primary operational data. They represent elapsed time since some fixed epoch (default is J2000, or 12:00 noon TT on January 1, 2000). This is the same instant in time across the different clocks; the difference between the *trunc_julian_time* representation and the *seconds* representation is best seen in this example:

> Terrestrial Time (TT) ticks in lockstep with atomic time (TAI), but they are offset by a constant value. The J2000 epoch occurred at the same instant for both, so *seconds* (since J2000) will be the same for both since they tick in lockstep. Conversely, the Truncated Julian Time "epochs" are clock-dependent, so they occurred at different instants. Since TT and TAI are offset from each other, their *trunc_julian_time* values will always differ by a constant amount.

The epoch can be easily configured to any desired time, J2000 was chosen simply because it is a well known data point. Setting the epoch closer to the simulation time would give a smaller value on the elapsed time, allowing for more precise time interval specifications (using J2000 as the epoch does allow microsecond significance, which is adequate for most purposes). Note that changing the epoch WILL NOT affect the precision of absolute time, since that will still be based on Truncated Julian Time (currently with significance of $O(10^{-5})s$). In contrast, the *julian_date* variable has only millisecond significance.

To anchor the operational values of *seconds* and *days* to a fixed time, the value *tjt_at_epoch* provides the Truncated Julian Time at the instant that *seconds = 0.0.*

While seconds and days are trivially incremented throughout the simulation, the calendar variables require a more time-consuming update, and so are updated only as necessary. The simulation developer may (or may not) have scheduled these, and some time representations may have calendars that are updated regularly, some rarely, some never. To check how often those updates are scheduled, look at the S_define file for the statements similar to:

```
{DYNAMICS, environment) environment/time: time.utc.calendar_update(
        In  double  simtime =  sys.exec.out.time);
```

Note that the simtime argument is used for the purposes of verifying that the update is actually needed (occasionally, calendar updates are called behind the scenes by other routines, so if time has not advanced since it was last updated, it will not be updated again). The actual time data is pulled directly from the time object, in this case, *utc.*

4.1.2.d User-Defined-Epoch Times (TimeUDE)

Again, these inherit from Time, but add the following variables:

- *epoch_year*
- *epoch_month*
- *epoch_day*
- *epoch_hour*
- *epoch_minute*
- *epoch_second*
- *clock_day*
- *clock_hour*
- *clock_minute*
- *clock_second*
- *epoch_format*
- *initial_value_format*
- *epoch_defined_in_name*

The *epoch_*** values are the values of the clock (declared by name in *epoch_defined_in_name*) at which the UDE time starts (i.e. has value = 0.0).

The *clock_*** values function comparably to the *calendar_*** variables found in the Standard Times; they provide the dd::hh::mm::ss clock value of the current time, measured in time elapsed since epoch. They may also be used to set the epoch of one UDE by specifying it in terms of the *clock_*** value of another UDE.

Note that UDE times do not include year and month, since these are not well-defined quantities outside of a standard time reference (a day is 86,400 seconds, while a month is some variable number of days).

4.1.3 Initializing the Simulation

In any application where specific absolute times are needed (e.g., ephemeris), the simulation must be initialized to some known value. The start of the simulation always has a Dynamic Time value of 0.0, but may have a UTC value of 2005/Jan/23::15:30:05 (as an example). Initialization in this context means setting the starting times correctly. This section describes how to do that.

In very basic simulations, there may be no need for an absolute reference time, nor a second clock. In that case no additional times should be declared in the S_define, and only *dyn-time* used. Since *dyn-time* will always start running at 0.0, that eliminates the need for an initialization process. The code will terminate if the user includes any additional clocks (beyond the default Dynamic Time) but fails to specify a method for initializing the simulation.

Where initialization is required, those values are set in an input file or modified-data file, discussed below.

4.1.3.a Initialization requirements

The following is a list of requirements pertinent to initialization. This is provided as a reference so that the user can identify the capability of the Time Representations Model.

- The initialization is available from any Standard or UDE time.
- The initialization time can be expressed in any of the following formats:
  - Gregorian calendar (Standard Times only)
  - dd::hh::mm::ss clock format (UDE Times only)
  - Julian, Modified Julian, or Truncated Julian formats (Standard Times only)
  - days since epoch
  - seconds since epoch
- Initializing a UDE Time (such as Mission-Elapsed-Time) requires that the UDE be anchored to another defined time representation. The following options are available:

- Given an absolute UDE epoch time (defined with respect to some other Derived Time (e.g. 0.0 UDE (tjt)= 123.456 TAI (tjt))) and an absolute start time from some Derived Time of known epoch (e.g. sim_start = 123.456 UTC (tjt)), then the initial value of the UDE time can be calculated.
- Given an absolute UDE epoch time and UDE initial value, then the absolute start time can be calculated and used to initialize other times, and the simulation.
- Given an absolute simulation-start time from some Derived Time and an initial value for the UDE time, then the absolute UDE epoch time (mission start time) can be determined.

4.1.4 Input Files

4.1.4.a Defining the initialization time with a Standard Time

If some real time is to be used to start the simulation, it must be expressed in some representation (e.g. UTC). The first value to specify is that representation, and it must be specified by name.

```
time.time_manager_init.initializer ="UTC";
```

Next, the format in which the time is expressed must be declared. This could be Julian, modified_julian, truncated_julian, or calendar. Obviously, the choice is determined by the data available to the user. The choice must be preceded by "TimeEnum::" as in the example below.

```
time.time_manager_init.sim_start_format = TimeEnum::calendar;
```

In this case, calendar was chosen, it is now necessary to fully define the mission start time in a calendar format on a UTC clock.

```
time.utc.calendar_year = 1998;
time.utc.calendar_month = 12;
time.utc.calendar_day = 31;
time.utc.calendar_hour = 23;
time.utc.calendar_minute = 59;
time.utc.calendar_second = 50.0;
```

If the start time were known in a Truncated Julian format on the UT1 clock, for example, this may have looked like:

```
time.manager_init.initializer = "UT1";
time.manager_init.sim_start_format = TimeEnum::truncated_julian;
time.ut1.trunc_julian_time = 12345.6789;
```

The code has some internal verification processes that will catch most of the common errors that users are likely to make in defining these values, but it is always better to get it right the first time.

4.1.4.b Initializing a UDE or MET / Setting the Epoch for a UDE or MET

There are two values that all clocks take at simulation-initialization:

- A definition of its epoch (the time at which it had a value of 0.0)
- Its value at the start of the simulation.

Standard clocks have a default epoch already defined. Dynamic Time has an automatic epoch of "now". User-Defined-Epoch clocks (UDE and the more flexible MET clocks) need additional definition.

Unless the UDE is being used to initialize the simulation, it is necessary that only one of these is defined. The other will be calculated. Specifying both will overconstrain the system and lead to a failure. If the UDE is being used to initialize the simualtion, then both must be specified.

There are multiple ways to directly specify the epoch of a UDE, including:

- as a calendar date and time in some standard clock,
- as a Julian, Modified Julian, or Truncated Julian value in some standard clock,
- as some number of days or seconds elapsed since, or preceding, the epoch of another clock (Standard or another UDE)
- as some number of days or seconds elapsed since, or preceding, the start of the simulation
- as some time elapsed since, or preceding, the epoch of another UDE, expressed in a clock format – days, hours, minutes and seconds.

The sequence for setting the epoch is as follows:

1. Decide on the clock to be used to define the epoch.
   - Set the variable *epoch_defined_in_name* to the name of the chosen clock. For example,

     ```
     jeod_time.ude_clock.epoch_defined_in_name = "UTC"
     ```

   - If setting it relative to simulation-start, use Dynamic Time ("Dyn").
2. Decide on which option to use to numerically define the epoch.
   - Set the value *epoch_format* accordingly to one of
     - *Julian* or *julian*
     - *modified_julian*
     - *truncated_julian*
     - *calendar*
     - *clock*
     - *days_since_epoch*

  - *seconds_since_epoch*
- Note that not all options are available for all clocks.
  - If setting relative to Dyn, use only *days_since_epoch* or *seconds_since_epoch*.
  - If setting relative to another UDE, use only *clock*, *days_since_epoch*, or *seconds_since_epoch*.
  - If setting relative to a standard clock, use any option other than *clock*.
- Prepend your selection with *trick.TimeEnum.*. For example, to specify the epoch in a UTC calendar:

```
jeod_time.ude_clock.epoch_format = trick.TimeEnum.calendar
```

3. Set the numerical value(s) appropriate to your choice.
   - If using *calendar*, assign values to *epoch_year, epoch_month, epoch_day, epoch_hour, epoch_minute,* and *epoch_second*.
   - If using *clock*, assign values to *clock_day, clock_hour, clock_minute,* and *clock_second*.
   - If using any of the other options, assign a value to *epoch_initializing_value*. This value is a dimensionless variable adopting units appropriate to its usage (i.e. days for the Julian and days-since-epoch options, and seconds for the seconds-since-epoch option).
   - The *epoch_**** and *clock_**** variables can be set directly. For example:

```
jeod_time.ude_clock.epoch_year = 1998
jeod_time.ude_clock.epoch_month = 12
jeod_time.ude_clock.epoch_day = 31
jeod_time.ude_clock.epoch_hour = 23
jeod_time.ude_clock.epoch_minute = 59
jeod_time.ude_clock.epoch_second = 55.0
```

   - The *epoch_initializing_value* must be set with a call using the format:

```
jeod_time.ude_clock.set_epoch_initializing_value(0.0,<value>)
```

     The first argument must be 0.0, the second argument is the assigned value.

Alternatively, the epoch may be inferred by initializing the simulation with respect to one clock, and setting the corresponding initial value of the UDE. The variable *initial_value_format* specifies which data set to look for in setting the initial value. The options are:

- *clock*
- *days_since_epoch*
- *seconds_since_epoch*

If using *clock*, assign values to *clock_day, clock_hour, clock_minute,* and *clock_second*. If using either of the other options, assign a value to *initializing_value*. Like *epoch_initializing_value*, this value is a dimensionless variable adopting units appropriate to its usage (i.e. days for the days-since-epoch option, and seconds for the seconds-since-epoch option). The *clock_**** and *initializing_value* variables may be set directly. For example:

```
jeod_time.ude_clock.initial_value_format = trick.TimeEnum.seconds_since_epoch
jeod_time.ude_clock.initializing_value = -5.0
```

This will start the clock with a value of -5.0 seconds, thereby placing its epoch 5.0 seconds into the simulation. Equivalently, the epoch could have been set to be 5.0 seconds after the simulation start-time. The two examples presented here – for setting the epoch and for setting the initial time – are equivalent when used with the example in the previous section on initializing the simulation from the UTC calendar representation.

Note that defining initial values for both *initializing_value* and any of the *clock_*** variables will set up an internal conflict and the code will fail.

4.1.4.c Initializing the simulation with a UDE or MET time

To initialize the simulation with a UDE or MET time, that time must have an epoch defined in some other time-type, and have an initial value with respect to that epoch. This is best seen in an example, such as at *verif/SIM_5_all_inclusive/SET_test/RUN_UDE_initialized/input.py*:

```
time.manager_init.initializer = "met_veh1"

time.metveh1.epoch_defined_in_name = "UTC"
time.metveh1.epoch_format = TimeEnum::calendar
time.metveh1.epoch_year = 1998
time.metveh1.epoch_month = 12
time.metveh1.epoch_day = 31
time.metveh1.epoch_hour = 23
time.metveh1.epoch_minute = 59
time.metveh1.epoch_second = 0.0

#////////////////////////////////////////////////////////////////////
#// These two blocks are equivalent
#////////////////////////////////////////////////////////////////////
time.metveh1.clock_day = 0
time.metveh1.clock_hour = 0
time.metveh1.clock_minute = 0
time.metveh1.clock_second = 50.0
#////////////////////////////////////////////////////////////////////
#//time.metveh1.initial_value_format = TimeEnum::seconds_since_epoch
#//time.metveh1.initializing_value = 50.0
#////////////////////////////////////////////////////////////////////

...

time.metveh1.update_from_name = "TAI";
```

This example will start the *metveh1* clock at a time corresponding to 1998/12/31::23:59:0.0 UTC. The simulation will start when *metveh1* = 50.0 seconds and *metveh1* will tick with TAI. Note –

holds on MET times can only be processed following simulation start, so in this example, *metveh1* must have a value of 50.0 seconds at a time corresponding to 50.0 TAI-seconds after the epoch time.

Since TAI and UTC are in lock-step during this interval, this example is equivalent to initializing the simulation at 1998/12/31::23:59:50.0 UTC, and setting the initial value of *metveh1* to 50.0 seconds.

4.1.4.d Defining the Trees

The final part of the input file for time comprises setting up the initialization and update trees. Subject to the availability of converter functions, any time type can (in principle) be initialized or updated by any other time type. However, it is recommended that TAI be derived from Dynamic Time, and other types from TAI; in particular, due to the implementation of leap seconds, TAI is not uniquely identified from all UTC values, and TAI should never be updated from UTC.

```
time.time_tai.initialize_from_name ="UTC"
time.time_tai.update_from_name = "Dyn"
```

This states that the initial value of TAI is going to be set by the initial value of UTC (so there had better be a UTC to TAI converter available and declared in the S_define). Folowing commencement of the simulation, TAI will continue to get its values based on the value of Dynamic Time (*manager.dyn_time*). If the user omits any part of the tree specification, the code has sufficient capability that it can search for the most direct path to provide a means of initializing or updating a time type that has no specified path. If such a path is found, a low-priority message will be sent to the message handler to alert the user to the paths selected. The code will terminate immediately if any of the following conditions are met:

- The user did not specify a full tree, and the auto-generator could not find a suitable path.
- The user sets up these paths in such a way that they generate circular dependency (e.g. type XXX updates from YYY, and YYY from XXX).
- A path is specified but there is no converter to handle the conversion.
- The specified initialization path does not form a continuous connection from the initializer to each of the time types.
- The specified update path does not form a continuous connection from Dynamic Time to each of the time types.
- The *initializer* has a defined value for its *initialize_from_name* variable (the *initializer* gets its initial data from user-input, not from another clock) [1].

[1] If editing an existing simulation to use a different clock as the *initializer*, overwrite the new *initializer*'s *initialize_from_name* so that it is empty (“”), and define a value for *initialize_from_name* for the old *initializer*.

4.1.4.e Commanding changes mid-simulation

The *manager.update* routine checks the current Simulator Time with its recording of the Simulator Time the last time the Time Manager was called. The user may specify changes to various values (e.g., start and stop a MET, change the scale-factor on the Dynamic Time) based on particular events, or on a predetermined Simulator Time-based schedule. While those changes are implemented before the call is made to the *manager.update* routine for a given Simulator Time, this is insufficient to ensure that the *manager.update* routine is run with those changes set.

The problem lies in the behind-the-scenes interaction between the dynamic integrator and the Time Manager. With some integrators (e.g. Runge-Kutta 4), the time must be evaluated at the end of the previous step as a component of the previous integration. So at step *i*, the value for time is needed at *i, i+0.5,* and *i+1.* Once step *i* is complete, changes made for step *i+1* are implemented, and step *i+1* begins; however, time has already been calculated at *i+1*, and the update will not be called again for this instant of time without further intervention.

Therefore, it is recommended that whenever changes are being commanded, that the Time Manager's recorded value of Simulator Time be manipulated to ensure that when the *manager.update* routine is called, the current Simulator Time will not agree with the recorded value. A safe value is 0, since Simulator Time starts at 0 and always increments.

```
read = 10
time.metveh2.hold = true
time.manager.simtime = 0
```

4.1.4.f Turning off Warnings

When setting the time, it is easy to overlook a critical step and finish with an unrealistic time. Fortunately, the most obvious errors are easily caught at initialization by testing the value of Truncated Julian Time. If it has a value of zero or less, a warning will be sent to the Message Handler. Developers of those historical simulations for which Truncated Julian Time should be less than zero (i.e. pre-May 24, 1968) have an option to disable this warning message by setting the flag *send_warning_pre_1968 = false.*

4.2 Integration

This section is intended for users who are incorporating the Time Representations Model into a simulation. It contains the basic requirements and step-by-step example of the construction of the *Trick S_define* file. For more thorough examples, the verif directory in the standard release of JEOD v5.1 has a "tutorial" feel to it, stepping through from simple cases to more complex examples as it progresses from *SIM_1_dyn_only* through *SIM_5_all_inclusive*.

4.2.1 S_define requirements

The time object in the S_define is structured with the following elements:

- Declare the following instances:
  - TimeManager
  - TimeManagerInit
  - Desired time types (NOTE: DynTime is automatically declared within the TimeManager; declaring an instance of DynTime could cause a failure when the code is confused about which version of DynTime to use).
  - Desired time converters. Note that converters may be bidirectional, and that the code is sufficiently capable of determining the appropriate direction. This is important for the user because the labels may be confusing; for example, the UTC to TAI converter is labeled TAI_UTC. In the case that the converter is desired in one direction at initialization, and the other during simulation, IT IS NOT necessary to declare it twice.
- Call the following initialization-class functions:
  - The registration function for each of the declared time types to register them with the Time Manager. This function adds the time type to the Manager's list, stores its location in that list as a value in the time type itself, and stores the address of the manager in each time type.
  - The registration functions for the declared time converters to register them with the Time Manager. This function adds the converter to the Manager's list of available converters.
  - The Time Manager initialize function (NOTE – this is a small function that calls the main initialization functions in TimeManagerInit).
- Call the following run-time functions:
  - The update function is called at any desired rate. This will update the decimal counter-type representation, ***but not the calendar representation***, of all declared time-types.
  - *(Optional)* the calendar-update function on any desired time-types at any desired rate. This function will convert the decimal counter-type representation to a date::hr:min:sec representation. Each call to calendar_update is type-specific and will update one, ***and***

only one, time-type. Any call to the calendar-update will run the full decimal update method for all time-types if it has not already been done at that particular Simulator Time. Consequently, the call to the regular update method is redundant if a calendar update is sequenced at the same rate.

4.2.2 S_define example

1. First, declare the *TimeManager* and *TimeManagerInit* classes. These must be declared for all simulations.

```
class TimeSimObject : public Trick::SimObject
{
   jeod::TimeManager     manager;
   jeod::TimeManagerInit manager_init;

   ...

};
```

2. Next, in this example, I declare two additional time types, and two time converters (there must be a converter from *dyn*, and at least one per declared time class. In this case, the conversion path may be *dyn* to *tai* to *utc,* it will not be possible to go from *dyn* to *utc* directly with these converters).

```
class TimeSimObject : public Trick::SimObject
{
   jeod::TimeManager     manager;
   jeod::TimeManagerInit manager_init;

   TimeTAI tai;
   TimeUTC utc;
   TimeConverter_Dyn_TAI  converter_dyn_tai;
   TimeConverter_TAI_UTC  converter_tai_utc;

   ...

};
```

3. Now I must register each of those types and their respective converters. The order in which this is done is not relevant; I chose to register the time type, then immediately register the converter I plan to use for that time type; that order is just to keep track and ensure that I did not overlook something.

```
class TimeSimObject : public Trick::SimObject
{
```

```
    jeod::TimeManager     manager;
    jeod::TimeManagerInit manager_init;

    TimeTAI tai;
    TimeUTC utc;
    TimeConverter_Dyn_TAI  converter_dyn_tai;
    TimeConverter_TAI_UTC  converter_tai_utc;

    TimeSimObject()
    {
        // Initialization jobs
        P_TIME ("initialization") manager.register_time( tai );
        P_TIME ("initialization") manager.register_converter( converter_dyn_tai );
        P_TIME ("initialization") manager.register_time( utc );
        P_TIME ("initialization") manager.register_converter( converter_tai_utc );

        ...

    }
};
```

4. Now initialize the manager. This must be done for all configurations.

```
class TimeSimObject : public Trick::SimObject
{
    jeod::TimeManager     manager;
    jeod::TimeManagerInit manager_init;

    TimeTAI tai;
    TimeUTC utc;
    TimeConverter_Dyn_TAI  converter_dyn_tai;
    TimeConverter_TAI_UTC  converter_tai_utc;

    TimeSimObject()
    {
        // Initialization jobs
        P_TIME ("initialization") manager.register_time( tai );
        P_TIME ("initialization") manager.register_converter( converter_dyn_tai );
        P_TIME ("initialization") manager.register_time( utc );
        P_TIME ("initialization") manager.register_converter( converter_tai_utc );

        P_TIME ("initialization") manager.initialize( &manager_init );

        ...

    }
};
```

5. Next, schedule the regular updates. This, also, is required. It will update *dyn*, *tai* and *utc*.

```
class TimeSimObject : public Trick::SimObject
{
   jeod::TimeManager     manager;
   jeod::TimeManagerInit manager_init;

   TimeTAI tai;
   TimeUTC utc;
   TimeConverter_Dyn_TAI  converter_dyn_tai;
   TimeConverter_TAI_UTC  converter_tai_utc;

   TimeSimObject()
   {
       // Initialization jobs
       P_TIME ("initialization") manager.register_time( tai );
       P_TIME ("initialization") manager.register_converter( converter_dyn_tai );
       P_TIME ("initialization") manager.register_time( utc );
       P_TIME ("initialization") manager.register_converter( converter_tai_utc );

       P_TIME ("initialization") manager.initialize( &manager_init );

       // Scheduled Jobs
       (DYNAMICS, "environment") manager.update( exec_get_sim_time() );
       ...
   }
};
```

6. Finally, I decided that the output from *UTC* would be more useful for my particular purpose if it were expressed as a calendar format rather than a decimal format. Here, I run the calendar update at the same rate as the regular update, so that the calendar format is always up-to-date with the decimal format. Otherwise, the two values may represent different times.

```
class TimeSimObject : public Trick::SimObject
{
   jeod::TimeManager     manager;
   jeod::TimeManagerInit manager_init;

   TimeTAI tai;
   TimeUTC utc;
   TimeConverter_Dyn_TAI  converter_dyn_tai;
   TimeConverter_TAI_UTC  converter_tai_utc;

   TimeSimObject()
   {
       // Initialization jobs
```

```
        P_TIME ("initialization") manager.register_time( tai );
        P_TIME ("initialization") manager.register_converter( converter_dyn_tai );
        P_TIME ("initialization") manager.register_time( utc );
        P_TIME ("initialization") manager.register_converter( converter_tai_utc );

        P_TIME ("initialization") manager.initialize( &manager_init );

        // Scheduled Jobs
        (DYNAMICS, "environment") manager.update( exec_get_sim_time() );
        (DYNAMICS, "environment") utc.calendar_update( exec_get_sim_time() );
    }
};
```

4.2.3 Updating the Data Tables

Default data is provided for the offset between UTC and UT1, on a day-by-day basis, from 1962 to the time just prior to to current JEOD version release date. Data for the offset between TAI and UTC (the number of leap seconds) is also provided. Because the UTC-UT1 offset changes continuously, this can quickly become outdated; the TAI-UTC data can also become outdated, but this is only updated once every 6 months so is not such a problem. If the data available needs to be brought up-to-date, new data can be obtained from the International Earth Rotation and Reference Systems Service (IERS) webiste http://www.iers.org, which has links to Data/Products and Earth Orientation Data.

If the user chooses to obtain their own data, we suggest the following steps:

1. Download the latest EOP 14 C04 (IAU2000) data file from the IERS at:

   https://datacenter.iers.org/data/latestVersion/EOP_14_C04_IAU2000A_yearly_files.txt

   remove the file extension (*.txt*) from the file name and save it to the data directory (*models/environment/time/data*).

2. OPTIONAL: Edit the data file by removing lines until data from only the desired time span remain in the file. The initial 14 header lines must NOT be removed.

3. Check the most recent leap second information, available in Bulletin C, (which is also linked from the IERS website), or from numerous other readily available sources. Bulletin C is published every 6 months, and announces the current difference between UTC and TAI (i.e., the number of leap seconds).

   In the data directory (*models/environment/time/data*) is a Perl script, *parser.pl*. This script contains a list of when all of the leap seconds were implemented. Check the current leap second number against the final entry in that script, currently ending with the line

   ```
   [ 2457754.5 , 37.0 ]    # 2017 JAN  1
   ```

   (In this case, the final value is 37 seconds, implemented at 0:00:00 hours on Jan 1, 2017, which has a Julian Date of 2457754.5)

If Bulletin C has a UTC to TAI difference that does *not* match the final entry in this table, the table should be updated by adding the Julian Date at last change and the new corresponding value, to the end of the table in the script.

4. In the data directory (*environment/time/data*), run the parser script with the command

```
perl parser.pl filename
```

where filename is the previously saved DUT1 data file (i.e., eopc04_14_IAU2000.62-now).

5. Verify that whichever of the two files *tai_to_utc.cc* and *tai_to_ut1.cc* are needed have been written appropriately.

4.2.4 Overriding the Data Tables

In some situations, the user may wish to specify a particular value in place of using the data look-up. This may be particularly relevant when developing historical or futuristic missions for which data is not available, or for carrying out a simulation using very recent data without going to the trouble of updating the data tables. However, this is generally not recommended because it locks UT1 and/or UTC to TAI, and prevents the automatic updates that enhance the clock precision.

First, both the TAI-UTC and TAI-UT1 converter classes have a flag titled *override_data_table* that defaults to *false*. This flag must be reset to *true* to enforce the override data.

Next, both classes have a variable to accommodate the desired value. These are titled *leap_sec_override_val* and *tai_to_ut1_override_val* for classes TimeConverter_TAI_UTC and TimeConverter_TAI_UT1 respectively.

All of these values can be set in an input file, with entries such as:

```
time.tai_ut1.override_data_table = true
time.tai_ut1.tai_to_ut1_override_val = -32.469
```

Note that when overriding the data for generating UT1, the variable is *tai_to_ut1_override_val*, i.e the value for converting from TAI to UT1. This is NOT the same as DUT1, which is used for converting from UTC to UT1. To generate the necessary value, it is also necessary to subtract off from DUT1 the number of leap seconds (the number of leap seconds provides the conversion from UTC to TAI).

When overriding the data for generating UTC, the number of leap seconds is used, and is (at least as of the publication of this document) a positive number.

The astute reader may have noticed, and be contemplating why, there is symmetry in the setting of the flag (tai_utc and tai_ut1 have their respective flags named identically), but not in the setting of the value. The explanation, just for the sake of curiosity, is as follows: for UT1, the data value is simply named for the converter class in which it resides, *TimeConverter_TAI_UT1*, so the data represents the conversion from TAI to UT1; for UTC, the same notation could have been confusing, because the value of the TAI to UTC converter is the negative number of leap seconds. We felt that having the negative sign would cause more issues than it solved, and so renamed it to simply represent the number of leap seconds, or the UTC to TAI conversion.

4.3 Extension

Several new files will be needed when extending the capability of the Time Representations Model with new Standard Times (note that JEOD allows for an indefinite number of UDE Times, adding UDE times is as simple as adding them to the S_define, and initializing them).

Perhaps the easiest way to do this is to copy an existing file of comparable purpose, and replace all instances of the old type with the new type. Remember to do this for both lowercase and uppercase.

Then edit the new file as needed.

4.3.1 How to add a new time-type:

An example of adding a new time type is verified at *SIM_6_extension* (on page 92) in the Verification section of this document. The code to support this verification is found at *time/verif/SIM_6_extension*.

1. Add it to a header file in the format:

```
  class TimeABC : public TimeStandard
{
  TimeABC( );
  ~TimeABC( );
}
```

2. Make a constructor for it:

   The *name* is required, any other values are optional

```
  TimeABC::TimeABC()
  {
    name = "ABC";
    // optional content
    // e.g., set_epoch();
  }
```

3. Make a destructor for it

4. Add a new converter class(optional)

   Presumably, if this is a new time-type, a new converter will be needed. See the next section on how to add a new TimeConverter Class.

5. Add it to the S_define in 2 places

   First, declare it

```
  class YourTimeSimObject : public Trick::SimObject
  {
```

```
...

TimeABC  abc;

...

YourTimeSimObject()
{
   ...
   P_TIME ("initialization") time_manager.register_time( abc );
   ...
}
};
```

Second, register it.

```
P_TIME (initialization) environment/time:     time.manager.register_time(
         In        JeodBaseTime  & time_reg = time.abc);
```

6. Update input file (optional) Add values to the input file to describe the location in the initialization and update tree

4.3.2 How to Add a New TimeConverter Function

In some cases, the standard release of JEOD v5.1 is equipped with one-way converters, but the functionality was not provided for both directions. For example, it is possible to convert UT1 to GMST, but not vice versa. If there is need for the reverse functionality, that function can easily be added by editing two files.

4.3.2.a Change the Constructor

In the appropriate TimeConverter_xxx_yyy function, there are two areas to change. The first is in the constructor, where a pair of functionality flags will likely be set to False. Let us assume that the *a_to_b* converter is defined, and the *b_to_a* converter is only available at runtime. Then the following flags should be set:

```
TimeConverter_XXX_YYY::TimeConverter_XXX_YYY()
{
  valid_directions = A_TO_B | B_TO_A_UPDATE;
  // other code
}
```

4.3.2.b Add the function

Further down in this file, you will find the code for the defined converter function (*TimeConverter_XXX_YYY::convert_a_to_b()*),

and for the reverse function (*TimeConverter_XXX_YYY::convert_b_to_a()*).

4.3.2.c Declare the function

The default (inherited from *TimeConverter*) version of an undefined converter function is intended to cause execution to stop immediately. The appropriate header file must be changed to declare that this new function exists, and prevent inheritance of the termination version.

In file *include/time_converter_xxx_yyy.hh,* add the following line:

```
 // convert_b_to_a: Apply the converter in the reverse direction
    void convert_b_to_a(void);
```

The function is now ready to go; to use it, add it to the definition of the appropriate tree in the input file (see Analysis section of this User Guide).

4.3.3 How to Add a TimeConverter Class

These steps are for users intending to add completely new converter functionality

4.3.3.a Write a header file for the new class

In the format

```
class TimeConverter_XYZ_ABC : public TimeConverter {

   JEOD_MAKE_SIM_INTERFACES(TimeConverter_TAI_GPS)

// Member Data
private:
 TimeXYZ * xyz_ptr; /* **
  Converter parent time, always a TimeXYZ for this converter. */
 TimeABC * abc_ptr; /* **
  Converter parent time, always a TimeABC for this converter. */

// Member functions:
public:
// Constructor
  TimeConverter_TAI_GPS();
// Destructor
  ~TimeConverter_TAI_GPS();

private:
// Initialize the converter
  void initialize( JeodBaseTime * parent, JeodBaseTime * child, const int direction);
```

```
// convert_a_to_b: Apply the converter in the forward direction
  void convert_a_to_b(void);  // optional

// convert_b_to_a: Apply the converter in the reverse direction
  void convert_b_to_a(void);  // optional
}
```

Either or both of the converter functions may be included. Having neither is not illegal, but is redundant (the converter has no functionality). Having both is also usually redundant, unless one is intended for use during initialization and the other during updates.

4.3.3.b Write the source code:

The *initializer* function must be defined, along with *convert_a_to_b( )* and *convert_b_to_a( )* if included in the header file.

4.3.3.b.i Initializer The *initializer* sets the pointers and initializes any data values required in the conversion process. In the case that the conversion process is itself a function of time, it must be initialized with respect to some time. This function is called as needed during the initialization process: when the time-types are being initialized, or when the update tree is being generated. If a converter is needed for either of those tasks, this function is called. It can be assumed that the *parent* type has been initialized already, and that the time is known in that representation. It should be assumed that the *child* type has not. Because either type (*a* or *b,* i.e. *xyz* or *abc*) can be known a priori, there are two "directions" available.

A simple example may look like:

```
void TimeConverter_XYZ_ABC::initializer( // Return: -- Void
   JeodBaseTime * const parent_ptr,
   JeodBaseTime * const child_ptr,
   const int int_dir)
{
 verify_setup(parent_ptr,child_ptr,int_dir);

 if (int_dir == 1) {
   xyz_ptr = dynamic_cast<TimeXYZ *> (parent_ptr);
   abc_ptr = dynamic_cast<TimeABC *> (child_ptr);
 }

 else if (int_dir == -1) {
   xyz_ptr = dynamic_cast<TimeXYZ *> (child_ptr);
   abc_ptr = dynamic_cast<TimeABC *> (parent_ptr);
 }
 a_to_b_offset = some_value_or_function;
```

```
 initialized = true;
 return;
}
```

4.3.3.b.ii Converter functions The converter functions are described in more detail in the previous section, *How to Add a New Converter Function* (on page 75).

4.3.3.b.iii Constructor The master class, *TimeConverter*, sets *a_to_b_offset = 0.0*, and *initialized = false.* These values are inherited by default; *initialized* will be reset to t*rue* in the initialization function, and *a_to_b_offset* may be calculated in the initialization function if necessary. The functionality flags, *a_to_b_initialization, a_to_b_runtime, b_to_a_initialization,* and *b_to_a_runtime* should be set in the specific class constructor. The two time-type pointers should be initialized to NULL (they are reset in the initializer function). The identification names of the two time types between which this converter operates must be **identical to** the names of those time-types as they appear in their respective classes.

```
a_name = "XYZ";
b_name = "ABC";
```

4.3.3.b.iv Destructor Releasing the memory allocated in the name definition is no longer necessary in JEOD 4.0, as the a_name and b_name members have converted to STL strings.

4.3.3.c Add it to the S_define in 2 places

First, declare it. Second Register it.

```
class YourTimeSimObject : public Trick::SimObject
{

  ...

  TimeConverter_XYZ_ABC xyz_abc;

  ...

  YourTimeSimObject()
  {
     ...

     P_TIME ("initialization") manager.register_converter( xyz_abc );

     ...
  }
};
```

Chapter 5

Verification and Validation

5.1 Verification

5.1.1 Top-level requirement

Inspection time_1: Top-level inspection

This document structure, the code, and associated files have been inspected, and together satisfy requirement time_1.

5.1.2 Verification of Dynamic Time

Test time_1: SIM_1_dyn_only

Purpose:
To verify that the Dynamic Time functions as a stand-alone implementation of time that can be used for integration purposes.

Requirement:
Satisfactory conclusion of this test satisfies requirement time_7.

Satisfactory conclusion of this test, along with test time_2 satisfies requirement time_2

Procedure:
A simulation was created with no additional time registrations, and set to run such that Dynamic Time = Simulator Time. No specific time was set, and no initialization of any time-types was required.

Predictions:
Dynamic Time should progress with Simulator Time.

Results:
Dynamic Time progressed with Simulator Time.

Test time_2: SIM_2_dyn_plus_STD/RUN_scale_factor_changes/

Purpose:

1. To test whether Dynamic Time can also operate at some scaled version of Simulator Time.
2. To test whether that scale factor can be changed mid-simulation, and have Dynamic Time retain its smooth flow.

Requirement:

Satisfactory conclusion of this test, and that of time_1, satisfies requirement time_2.

Procedure:

A simulation was set running, in which Dynamic Time started synchronized to Simulator Time. At $SimulatorTime = 5$, the scale-factor was changed to -1.0 (corresponding to a time-reversal). At $SimulatorTime = 10$, the scale-factor was changed again, to 0.5. Then, at $SimulatorTime = 20$, the scale-factor was changed to -2.0. A Standard Time was also added to this simulation to ensure that the changes to Dynamic Time were propagated to Derived Times.

Predictions:

Dynamic Time should increase from 0 seconds to 5 seconds, then fall back to 0 seconds, then increase back to 5 seconds again (but over a longer Simulator Time interval), then decrease to -5 seconds before the simulation ends at $SimulatorTime = 25$. The Standard Time (in this case, TAI) should show the same pattern, only over a different range (Standard Times do not typically start at 0).

Results:

The following three graphs show the progression of Dynamic Time and TAI, as expected.

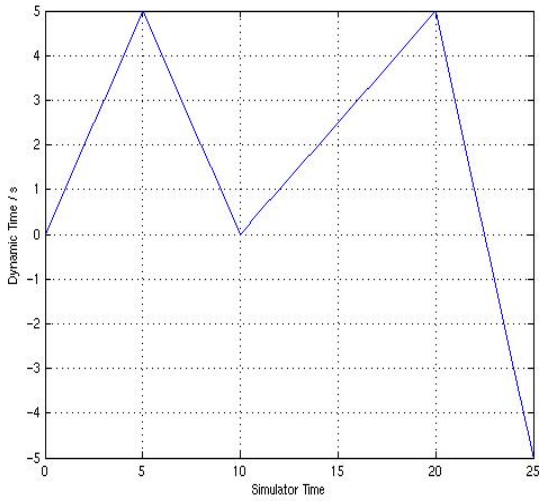

Figure 5.1: Variation with Simulator Time of Dynamic Time.

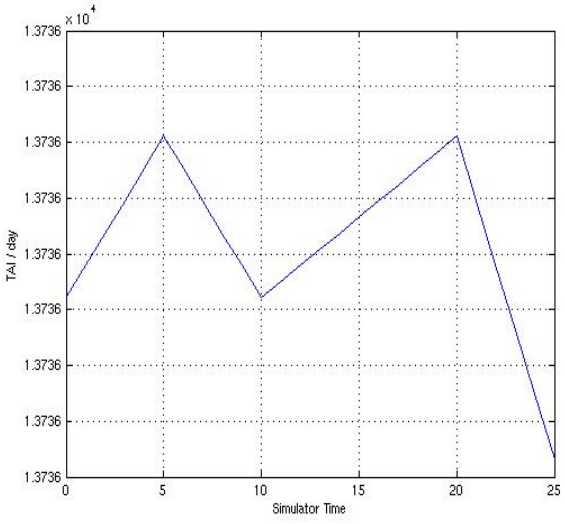

Figure 5.2: Variation with Simulator Time of TAI.

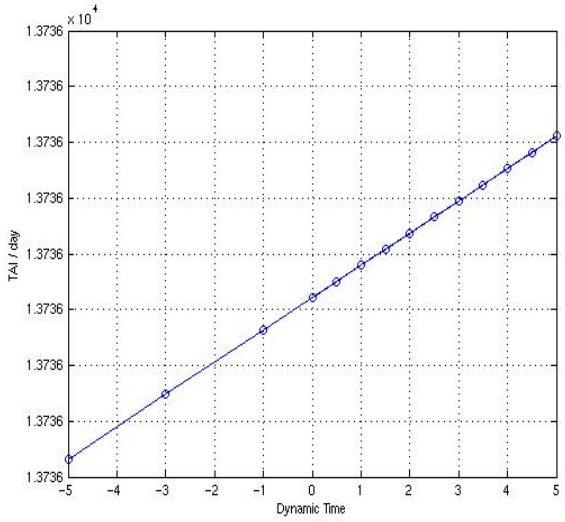

Figure 5.3: Variation with Dynamic Time of TAI.

5.1.3 Verification of Presence of Derived Times and of the Initialization and Propagation Thereof

Test time_3: SIM_5_all_inclusive/RUN_UTC_initialized

Purpose:
To ensure that the clocks specified in requirement time_3 are present and functioning.

Requirement:
Satisfactory conclusion of this test satisfies requirement time_3.

Satisfactory conclusion of this test, together with time_8 satisfies requirement time_8.

Procedure:
A simulation was developed that contained all clocks included in the release of JEOD v5.1.

Predictions:
All clocks will tick at their appropriate rates, with values consistent with each other.

Results:
All clocks performed as expected. Figure 5.4 shows the progression of the *calendar_second* and *clock_second* values during the simulation.

Some key observations from this plot:

- After kicking down to 0 seconds (at the end of the minute), UTC holds for 1 second. This is an effect of the leap second that was added there.
- MET1 is initialized in such a way that the respective values of UTC and MET1 are equivalent up to the addition of the leap second. Since MET1 is scheduled to update from TAI (and therefore ticks with TAI), it will not have a leap second and will be offset from UTC for all time after the leap second.
- MET2 is scheduled to start with a negative value, and to hold during the middle of the simulation.
- UT1 advances a little behind UTC; after the leap second, it is a little ahead.
- TDB is not shown; on this scale, it is an overlay of TT.
- Data is recorded every second; the last value recorded before the minute ticks over in UTC, UT1, and TT is the last value in the range [59,60) s, which is not consistent between clocks (as expected).

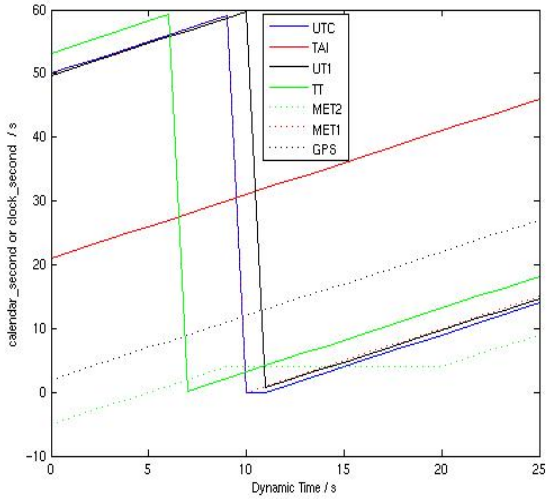

Figure 5.4: Variation with Dynamic Time of Derived Times.

5.1.3.a Verification of TDB to TAI Conversion

Test time_4: SIM_5_all_inclusive/RUN_UTC_initialized_tdb

Purpose:

To demonstrate that the newly added convert_b_to_a() function correctly implements the non-trivial conversion algorithm from TDB to TAI.

Requirement:

Satisfactory conclusion of this test satisfies requirement time_5, providing the ability for time clocks to be converted to a different type in the reverse direction.

Procedure:

The following entries in the input file were changed:

```
LOG_CYCLE = 1.0
execfile( "Log_data/log_rec_tdb.py" )

jeod_time.utc.calendar_year = 2017
jeod_time.utc.calendar_month = 10
jeod_time.utc.calendar_day = 28
jeod_time.utc.calendar_hour = 01.0
jeod_time.utc.calendar_minute = 23.0
jeod_time.utc.calendar_second = 45.0
```

to change the initialization time to match the initialization time of a results comparison python script.

```
jeod_time.tdb.initialize_from_name = "UTC"
```

to

```
jeod_time.tdb.initialize_from_name = "TAI"
```

to change from a UTC-initialized simulation to a TAI-initialized simulation.

```
trick.sim_services.exec_set_terminate_time(3600*2)
```

to allow the simulation to run for two hours.

The following changes were made in the *Log_data* directory.

Create a new *log_rec.py* file named *log_rec_tdb.py*. In this new file, remove all dr_group.add_variable("") lines of code, except for

```
dr_group.add_variable( "jeod_time.tai.calendar_second")
dr_group.add_variable( "jeod_time.tdb.calender_second")
```

and change these lines to

```
dr_group.add_variable( "jeod_time.tai.seconds")
dr_group.add-variable( "jeod_time.tdb.seconds")
```

This allows only the TAI and TDB variables to be recorded in seconds when the simulation runs.

Predictions:
All clocks will tick at their appropriate rates, with values consistent with each other. Results produced by the simulation and the comparison script should be equal.

Results:
All data consistent with expectations.

5.1.4 Verification of Initialization by Derived Times

5.1.4.a Verification of Initialization by Standard Time

Test time_5: SIM_5_all_inclusive

Purpose:
To demonstrate that changes to the identification of which type is the *initializer*, coupled with associated changes to the initialization and update trees, and declaration of appropriate initial values, will result in the simulation being initialized in whichever time-type is desired.

Requirement:
Satisfactory conclusion of this test, and that of time_8, satisfies requirement time_5.

Procedure:
The following entries in the input file were changed:

```
time.manager_init.initializer = "UTC";
time.utc.calendar_year = 1998;
  time.utc.calendar_month = 12;
  time.utc.calendar_day = 31;
  time.utc.calendar_hour = 23;
  time.utc.calendar_minute = 59;
  time.utc.calendar_second = 50.0;
```

by replacing reference to UTC with reference to the desired class.

The initialization and update tree definitions were changed, e.g. from

```
  time.tai.initialize_from_name = "UTC";
```

to

```
  time.utc.initialize_from_name = "TAI";
```

to change from a UTC-initialized simulation to a TAI-initialized simulation.

Predictions:

All clocks should still initialize, but with different values because the initialization time has changed (it is numerically the same, but now on a different clock). All clocks should progress as before.

Results:

All data was consistent with expectations.

Test time_6: SIM_2_dyn_plus_STD/initialize_by_value

Purpose:

To test the initialization routines, to ensure that a Standard Time can be initialized by a value expressed as a Truncated Julian Time value, a Modified Julian Time value, a Julian Day value, seconds since J2000, or days since J2000.

Requirement:

Satisfactory conclusion of this test, together with test time_7, and test time_8 satisfy requirements time_6 and time_4.

Procedure:

The input file for the run identified as *RUN_initialize_by_value* contains several lines of options. Each option was tested, one at a time, and the resulting data considered.

For each option, the simulation was initialized to a value of 10,000 (units correspond to the option). For each option, the output was plotted as a Truncated Julian Time representation.

Predictions:

- Truncated Julian Time

  The output should have a value of $10,000$ days.
- Modified Julian Time

  Modified Julian Time differs from truncated Julian Time by 40,000 days. If Modified Julian Time = $10,000$ days, then Truncated Julian Time = $-30,000$ days.
- Julian Time

  Julian Time differs from Truncated Julian Time by 2,440,000.5 days. If Julian Time = $10,000$ days, then Truncated Julian Time = -2.4300005×10^6 days.
- seconds_since_epoch

  The TAI epoch is set to J2000, corresponding to a Truncated Julian Time of 11544.4996275 days in TAI. $10,000$ seconds (or 0.11574 days) after this point gives Truncated Julian Time = 11544.61536824074 days.
- days_since_epoch

  10,000 days after the TAI epoch has a Truncated Julian Time of 21544.4996275 days.

Results:

All values agreed with their predicted values.

Test time_7: SIM_2_dyn_plus_STD/RUN_initialize_by_calendar

Purpose:
To test whether a Standard Time can be correctly initialized by a calendar representation.

Requirement:
Satisfactory conclusion of this test, together with test time_6, and test time_8 satisfy requirement time_6.

Procedure:
A simulation was created comprising Dynamic Time and TAI. The initialization was set to a calendar value of 2005/12/31::23:59:50.0 TAI.

Predictions:
The calendar value should correspond to a Truncated Julian Time of 13735.9998842593

Results:
The predicted value is correct.

5.1.4.b Verification of Initialization by User-defined-epoch Times

Test time_8: SIM_5_all_inclusive/RUN_UDE_initialized

Purpose:
To test whether the Time Representations Model can be initialized from a UDE time-type, using either an initial value, or a clock.

Requirement:
Satisfactory conclusion of this test, together with test time_7, and test time_6 satisfies requirement time_6.

Satisfactory conclusion of this test, together with test time_5, satisfies requirement time_5.

Satisfactory conclusion of this test, together with test time_3, satisfies requirement time_8.

Procedure:
In the all-inclusive simulation with an initialization by a UDE, there are two options for initializing the UDE (and hence the simulation). One is by using the initial value of the UDE in *seconds_since_epoch*, and the other by specifying a clock format (still representing time since epoch). The same simulation was run with both options.

Predictions:
The two runs should produce identical results, and the other time representations should generate data consistent with a simulation starting at 1998/12/31::23:59:50.0.

Results:
The data are as expected.

5.1.5 Verification of Data Over-rides

Test time_9: SIM_4_common_usage/RUN_JEOD1x_compatible

Purpose:

To ensure that the methods can be used for simulations set in the future for which data values are not included in the JEOD v5.1 release, and to demonstrate a method by which the JEOD v5.1 data output can be compared directly against that from previous releases.

Requirement:

Satisfactory conclusion of this test satisfies requirement time_9.

Procedure:

The flags *true_utc* and *true_ut1* were forced to false, effectively eliminating the capability of updating the offset values from the most recent data. Instead, an offset between, say TAI and UTC, will be calculated initially, and that value will remain as a constant for the duration of the simulation.

Predictions:

Running through the known instant of a leap second will produce no effect on UTC with the flags turned off. The two UT1 values (with flag set to true and false) will be identical at the start of the simulation, and gradually diverge throughout the simulation.

Results:

The data are as expected. In the following figure legends, "***-1" refers to the JEOD1.x compatible data, and "***-2" refers to the data obtained with the default settings in JEOD v5.1. Figure 5.5 shows the relation between TAI, UTC, and UT1 at a time near the end of the simulation.

Figure 5.6 represents a small part of figure 5.5, zoomed in. It shows the separation of the two UTC values at the point at which a leap second is encountered (Note that the leap second is added to the JEOD v5.1 version of UTC, and the JEOD1 version of UTC keeps ticking right through it).

Figure 5.7 represents a further zoom in, and shows the difference between the two UT1 times.

Finally, figure 5.8 shows the difference between the two UT1 values clearly growing over time. A similar plot differencing the two UTC values is not presented, it showed a flat line 0 preceding the leap second; thereafter, the difference was 1 second, except as the minutes ticked over, when it would jump to 59 seconds (because one time would tick over and go to zero while the other was still near 60).

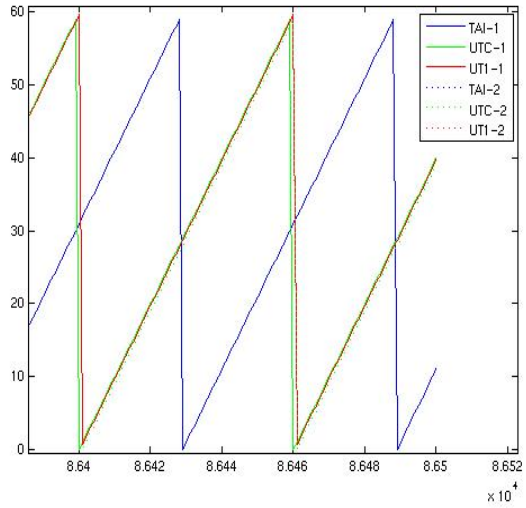

Figure 5.5: Showing the relation between TAI, UTC, and UT1.

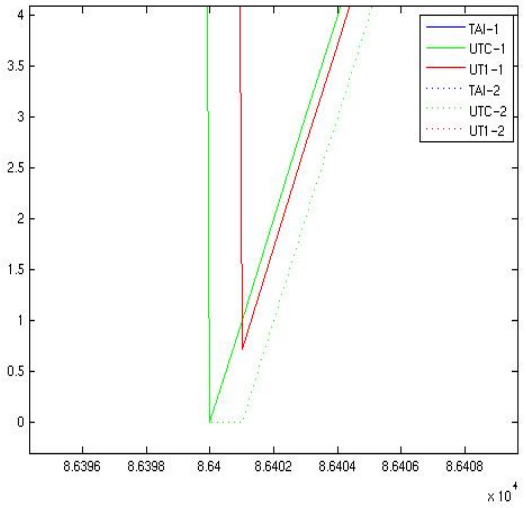

Figure 5.6: Values of UTC showing the addition of a leap second in the JEOD v5.1-compatible version, but not in previous versions.

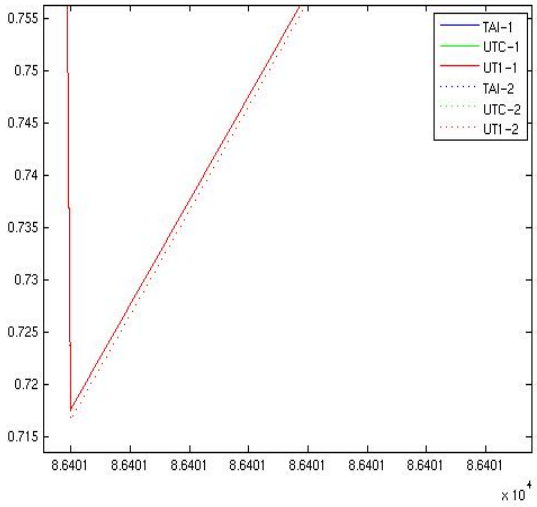

Figure 5.7: Values of UT1 showing the difference between performing regular updates and using a fixed data-file override value. This difference is after approximately 1 day of Dynamic Time.

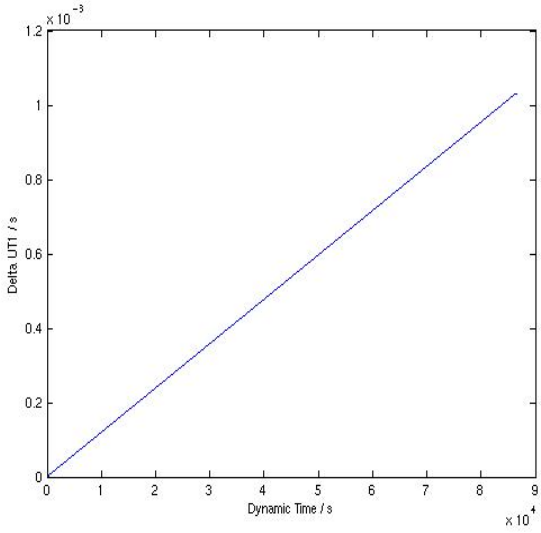

Figure 5.8: The difference,over a period of 1 day, between the values of UT1 using a regular update, versus using a fixed data-file override value.

5.1.6 Verification of Extension Capabilities

Test time_10: SIM_6_extension

Purpose:
To ensure that new clocks and converters can be added without altering the existing codebase.

Requirement:
Satisfactory conclusion of this test satisfies requirement time_10.

Procedure:
A new clock and accompanying converter were added to the simulation, as described in the *User Guide* (on page 74).

Predictions:
The time would evolve as Dynamic Time progresses.

Results:
The data are as expected.

5.2 Validation

The overall functionality of the Time Representations Model has been validated using respectable conversion tools, mostly provided by the United States' Naval Observatoy, to validate that our clocks are internally consistent, and that the decimal representation of time is consistent with the calendar/clock representation of time. In no test was a discrepancy found within the level of precision of those conversion tools.

Output has also been validated against output from earlier releases of JEOD; it has been demonstrated that for a JEOD 1.x-compatible simulation, flags can be set to provide equivalent data. However, it has been found that the new implementation is closer to clock standards than the old implementation; the old implementation should be used in very limited circumstances.

The time-reversal feature of the Time Representations Model has been validated internally, by comparing state data from forward and reverse simulations.

Test time_11: SIM_7_time_reversal

Purpose:
To evaluate the capability of the JEOD v5.1 models to handle time reversal simulations.

Requirements:
Satisfactory conclusion of this test satisfies requirement time_2.

Procedure:
A selection of the simulation runs from the top-level verification simulation, SIM_dyncomp were copied. These simulations were set to run forward for 60,000 seconds, at which time the time would reverse, and run backward for 60,000 seconds.

Predictions: The final state should be identical to the initial state. While this is not possible to achieve by any means other than a statistical fluke (due to numerical rounding), the extent to which the two states differ gives significant data on how well the time reversal functions.

Results:
For all tests, the same technique was used to graph the data, with all graphs showing the difference between the state in the forward and reverse parts of the simulation.

On the graph, time t=0 corresponds to the turnaround at simulation time = 60,000 seconds; there is no difference in the state at this time. Graphical time t= 60,000 seconds corresponds to the difference between the state at simulation time t=0 (in the forward component of the simulation), and the state at simulation time t= 120,000 seconds (in the reverse component of the simulation).

The following tests were used:

RUN_1: This run tests simple propagation in a spherical gravity field.

The position diverges by microns over the 120,000 seconds of the simulation (see figure 5.9).

The velocity diverged by nano-meters per second (see figure 5.10).

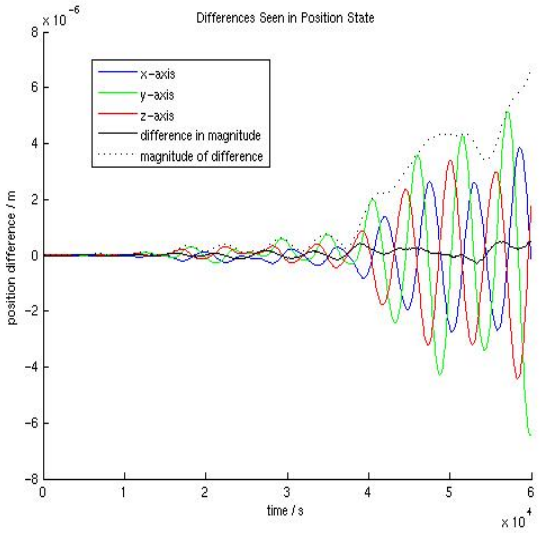

Figure 5.9: The difference in the position state for RUN_1.

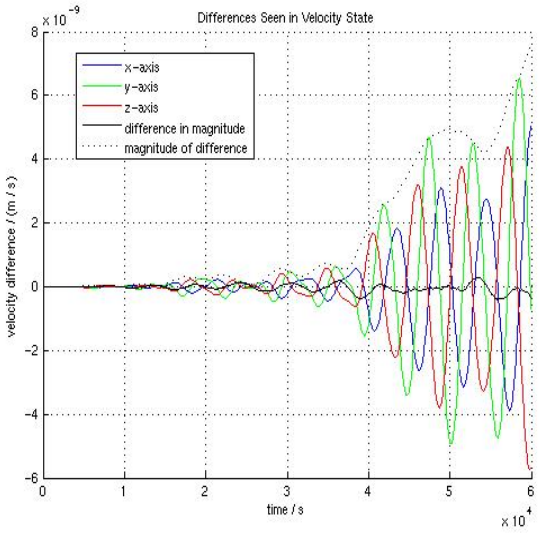

Figure 5.10: The difference in the velocity state for RUN_1.

RUN_3A This run tests simple propagation in a nonspherical, 4x4 gravity field.

The position diverges by centi-meters over the 120,000 seconds of the simulation (see figure 5.11).

The velocity diverged by $O(10^{-5})$ meters per second (see figure 5.12).

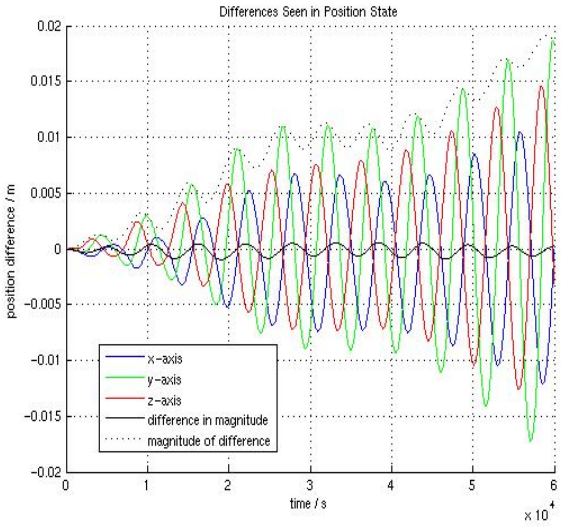

Figure 5.11: The difference in the position state for RUN_3A.

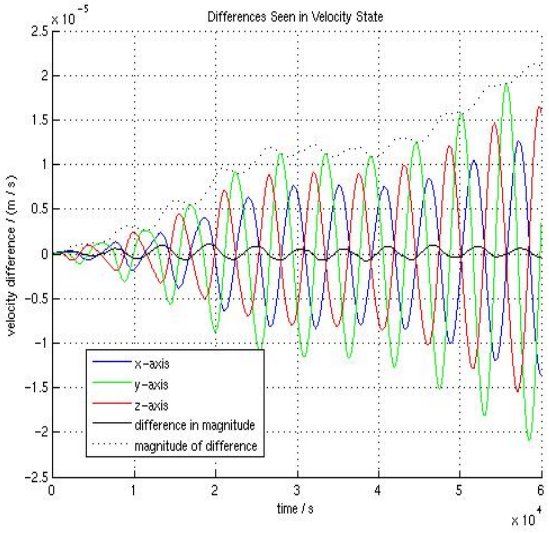

Figure 5.12: The difference in the velocity state for RUN_3A.

RUN_3B This run tests simple propagation in a nonspherical, 8x8 gravity field.

The position diverges by centimeters over the 120,000 seconds of the simulation (see figure 5.13).

The velocity diverged by $O(10^{-5})$ meters per second (see figure 5.14).

Because these data were significantly worse than the other simulations, we tried changing the rate at which the RNP matrix was being updated from once every 60 seconds to updates at 32 Hz, the same as the dynamic rate. The rationale was demonstrated to have some validity, although the divergences in the enhanced state were still relatively large (see figures 5.15 and 5.16 for the enhanced simulation data).

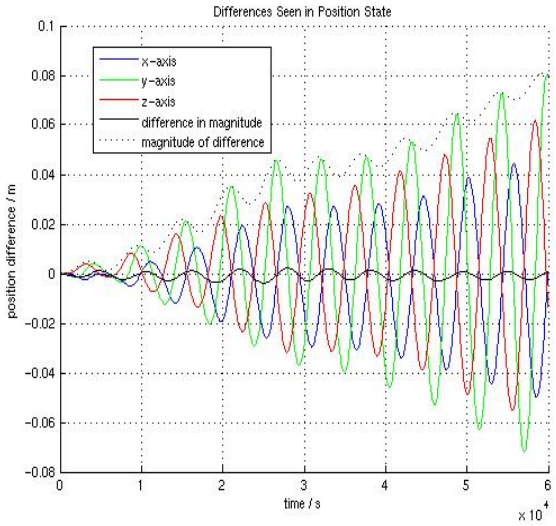

Figure 5.13: The difference in the position state for RUN_3B.

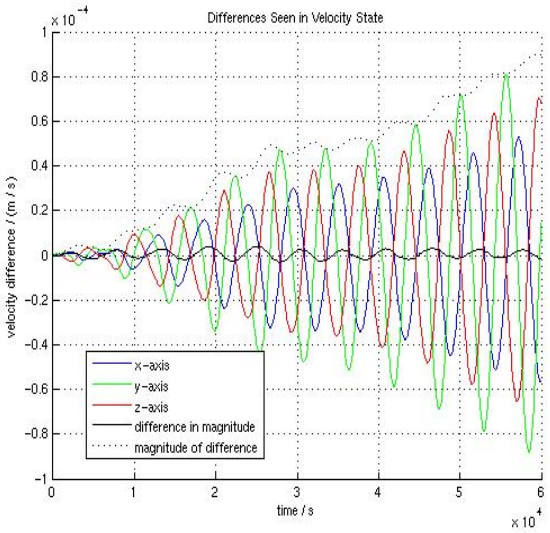

Figure 5.14: The difference in the velocity state for RUN_3B.

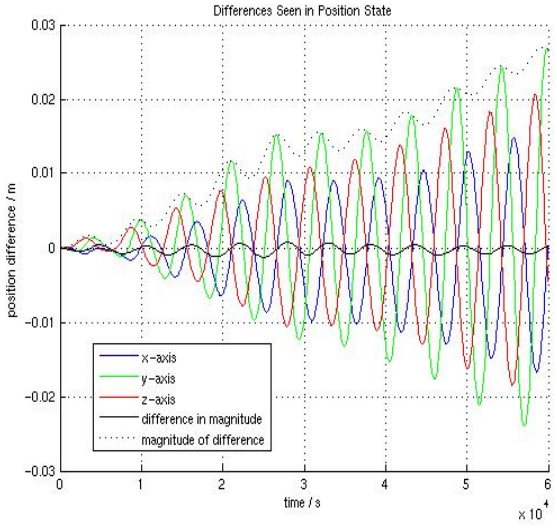

Figure 5.15: The difference in the position state for RUN_3B.

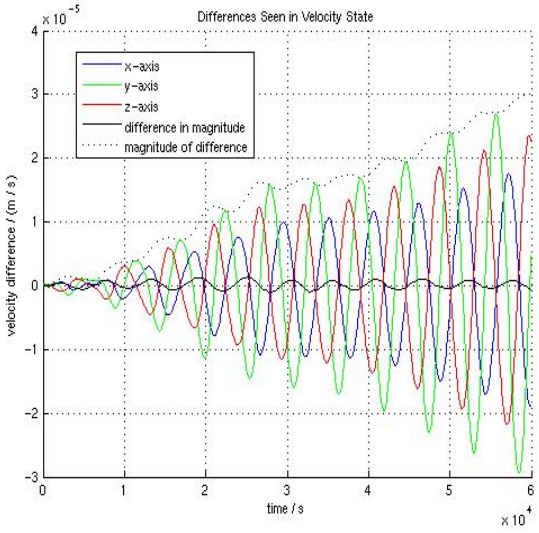

Figure 5.16: The difference in the velocity state for RUN_3B.

RUN_4 This run tests propagation in a spherical gravitational field with 3rd body perturbations.

The position diverges by $O(10^{-5})$ meters over the 120,000 seconds of the simulation (see figure 5.17).

The velocity diverged by $O(10^{-8})$ meters per second (see figure 5.18).

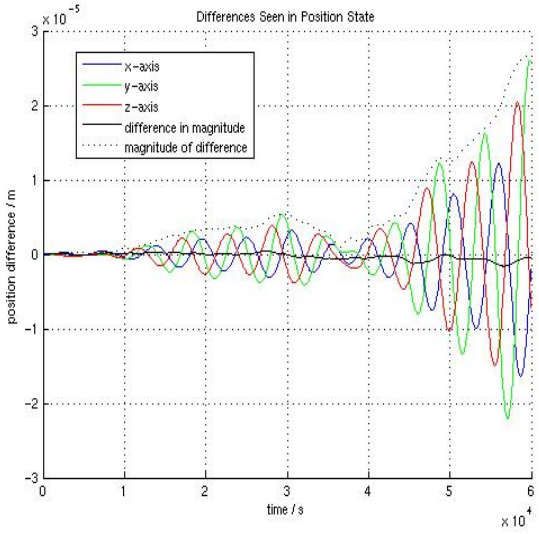

Figure 5.17: The difference in the position state for RUN_4.

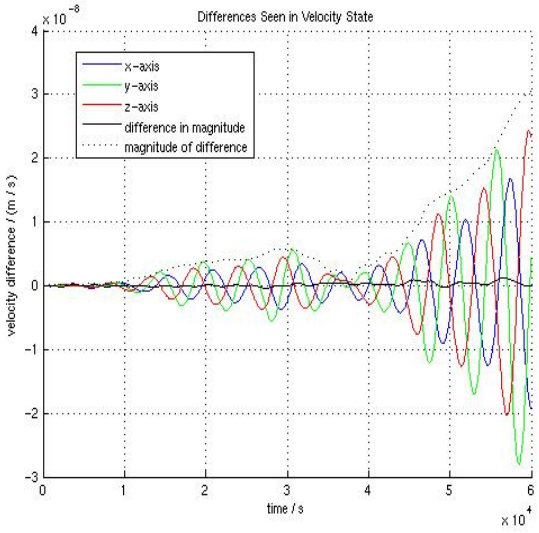

Figure 5.18: The difference in the velocity state for RUN_4.

RUN_6A This run tests propagation in a spherical gravitational field with aerodynamic drag effects included.

The position diverges by $O(10^{-4})$ meters over the 120,000 seconds of the simulation (see figure 5.19).

The velocity diverged by $O(10^{-7})$ meters per second (see figure 5.20).

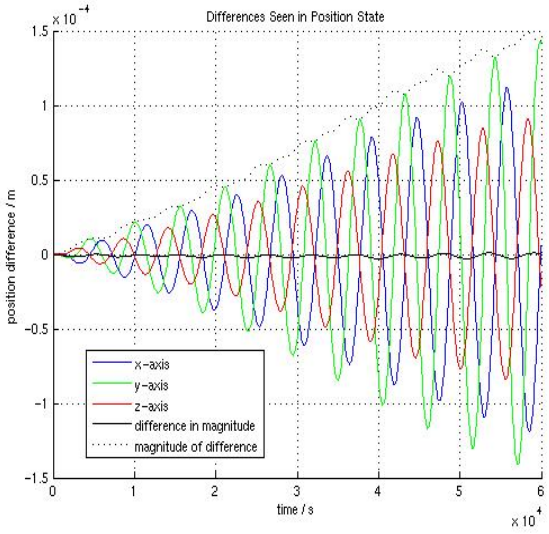

Figure 5.19: The difference in the position state for RUN_6A.

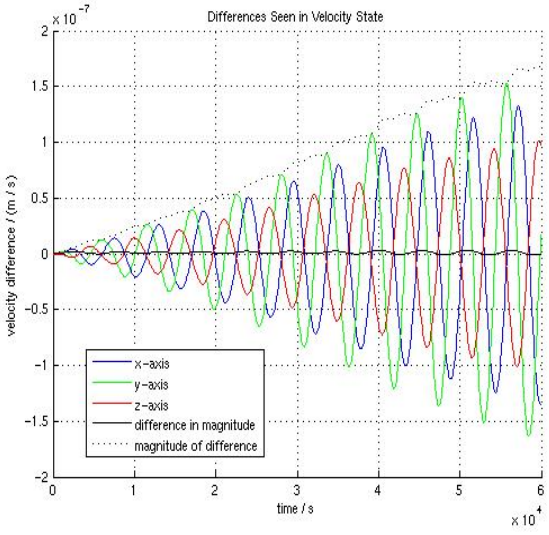

Figure 5.20: The difference in the velocity state for RUN_6A.

RUN_8B This run tests propagation in a spherical gravitational field with the rotational state also available. Only rotational data is provided here; the translational state is equivalent to that in RUN_1. RUN_8B was chosen because it starts with a non-zero rotational state.

The quaternion values diverge by $O(10^{-14})$ over the 120,000 seconds of the simulation (see figure 5.21).

The angular velocity diverged by $O(10^{-14})$ radians per second (see figure 5.22).

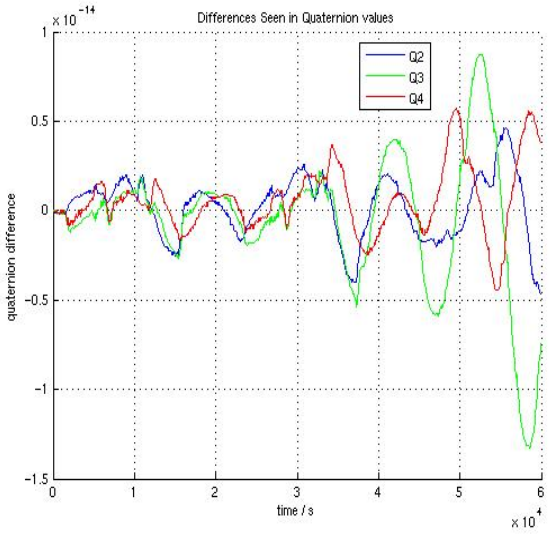

Figure 5.21: The difference in the quaternion values for RUN_8B.

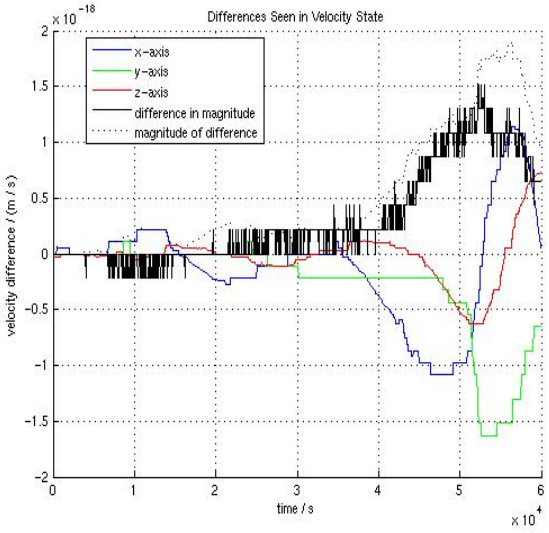

Figure 5.22: The difference in the angular velocity state for RUN_8B.

RUN_9D This run tests propagation in a spherical gravitational field with rotational state and external torques and forces. For this simulation, an external force and torque was applied between times t= 10,000 seconds and t = 20,000 seconds and then again between times t=100,000 seconds and t = 110,000 seconds to maintain the symmetry for the forward and reverse components.

The effect of the forces and torques on the state at the times for which they are active can be seen in figures 5.23 and 5.24.

The position diverges by $O(10^{-7})$ meters over the 120,000 seconds of the simulation (see figure 5.25).

The velocity diverged by $O(10^{-10})$ meters per second (see figure 5.26).

The quaternion values diverge by $O(10^{-14})$ over the 120,000 seconds of the simulation (see figure 5.27).

The angular velocity diverged by $O(10^{-19})$ radians per second, at the limit of resolution (see figure 5.28).

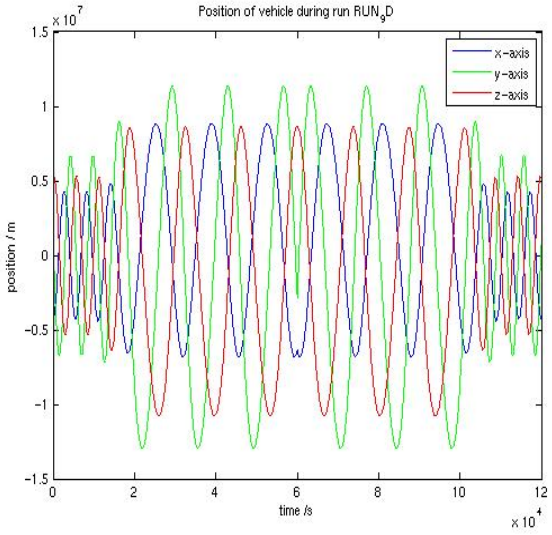

Figure 5.23: The position state for RUN_9D.

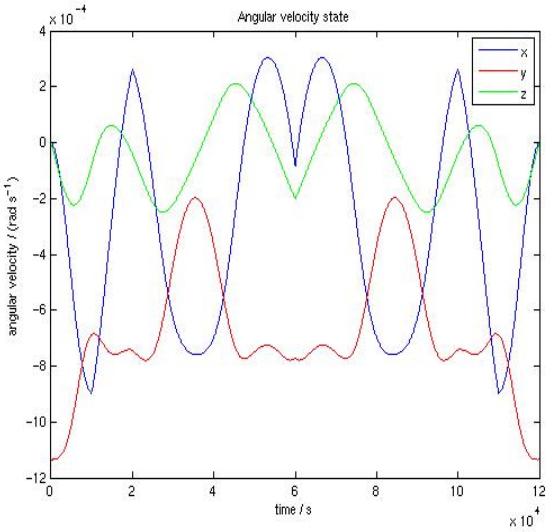

Figure 5.24: The angular velocity state for RUN_9D.

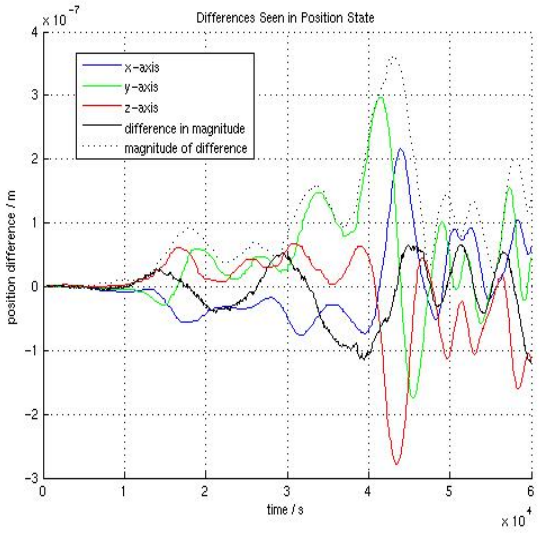

Figure 5.25: The difference in the position state for RUN_9D.

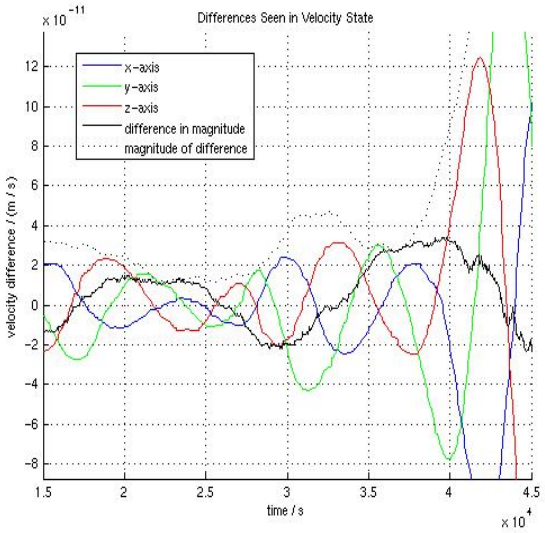

Figure 5.26: The difference in the velocity state for RUN_9D.

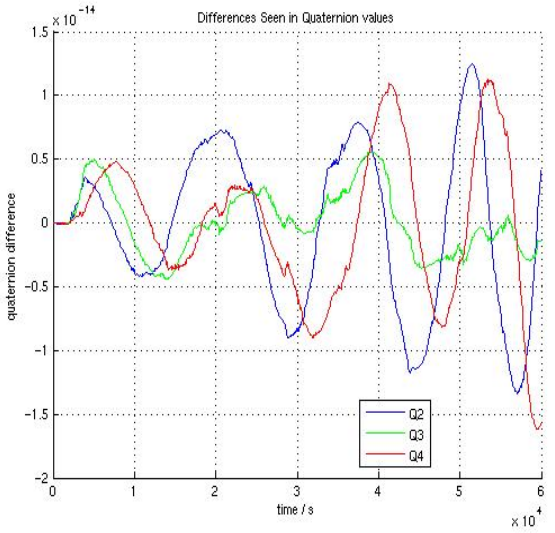

Figure 5.27: The difference in the quaternion values for RUN_9D.

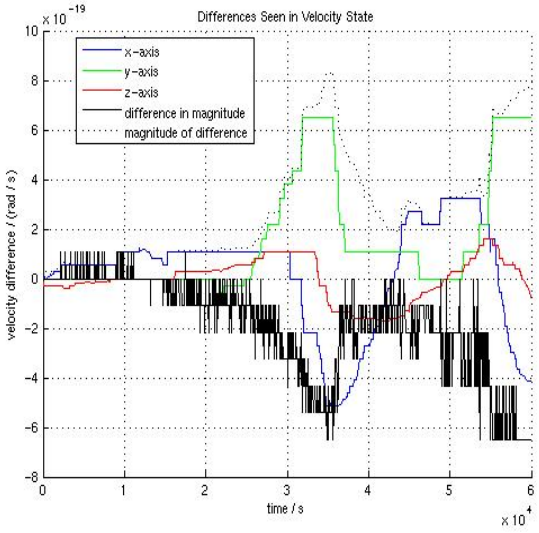

Figure 5.28: The difference in the angular velocity state for RUN_9D.

RUN_10A This final run tests propagation in a spherical gravitational field with gravity gradient torque present.

The position diverges by $O(10^{-6})$ meters over the 120,000 seconds of the simulation (see figure 5.29).

The velocity diverged by $O(10^{-9})$ meters per second (see figure 5.30).

The quaternion values diverge by $O(10^{-13})$ over the 120,000 seconds of the simulation (see figure 5.31).

The angular velocity diverged by $O(10^{-16})$ radians per second, at the limit of resolution (see figure 5.32).

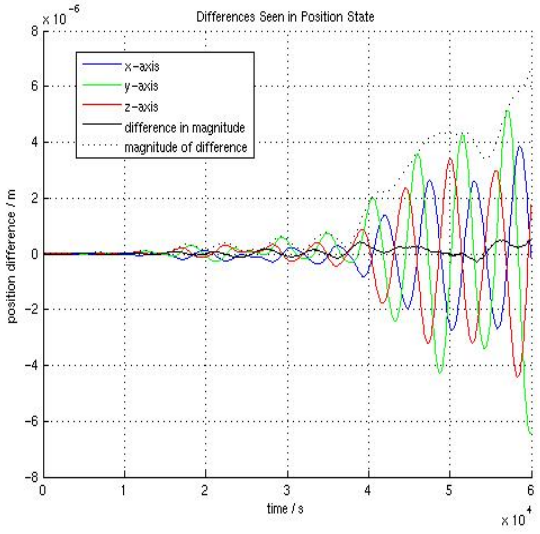

Figure 5.29: The difference in the position state for RUN_10A.

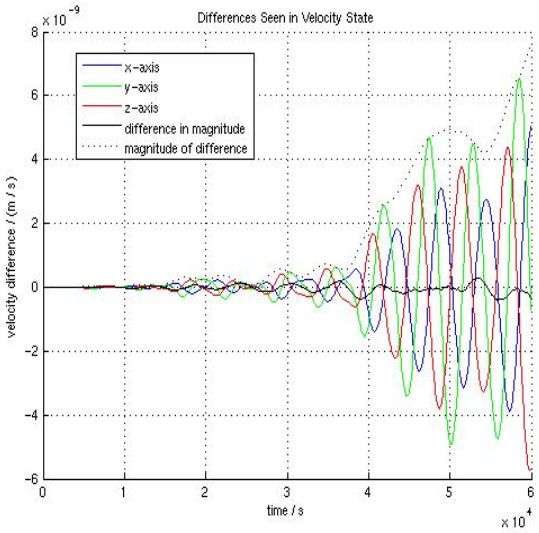

Figure 5.30: The difference in the velocity state for RUN_10A.

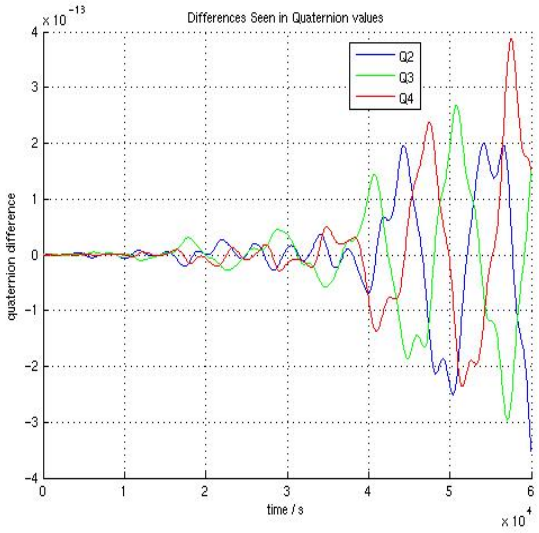

Figure 5.31: The difference in the quaternion values for RUN_10A.

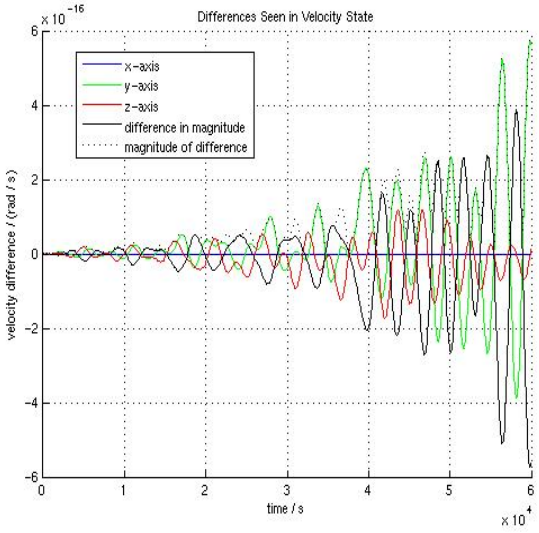

Figure 5.32: The difference in the angular velocity state for RUN_10A.

5.3 Metrics

5.3.1 Code Metrics

Table 5.1 presents coarse metrics on the source files that comprise the model.

Table 5.1: Coarse Metrics

File Name	Number of Lines			
	Blank	Comment	Code	Total
Total	0	0	0	0

Table 5.2 presents the extended cyclomatic complexity (ECC) of the methods defined in the model.

Table 5.2: Cyclomatic Complexity

Method	File	Line	ECC
jeod::JeodBaseTime::(std::set_ name (std::string name_in)	include/time.hh	190	1
jeod::JeodBaseTime::set_ index (int idx)	include/time.hh	198	1
jeod::JeodBaseTime::get_ index ()	include/time.hh	206	1
jeod::JeodBaseTime::override_ initialized (bool init)	include/time.hh	214	1
jeod::JeodBaseTime::is_ initialized ()	include/time.hh	222	1
jeod::TimeConverter:: override_initialized (bool init)	include/time_converter.hh	158	1
jeod::TimeConverter::get_a_ to_b_offset (void)	include/time_converter.hh	179	1
jeod::TimeConverter:: operator— (Time Converter::Direction a, TimeConverter::Direction b)	include/time_converter.hh	203	1
jeod::TimeGMST::set_epoch (void)	include/time_gmst.hh	101	1
jeod::JeodBaseTime::Jeod BaseTime (void)	src/time.cc	55	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodBaseTime::add_type_initialize (const int seeking_status JEOD_UNUSED, TimeManagerInit * time_manager_init JEOD_UNUSED)	src/time.cc	82	1
jeod::JeodBaseTime::add_parent (JeodBaseTime& parent)	src/time.cc	104	1
jeod::JeodBaseTime::initialize_from_parent (TimeManagerInit * time_manager_init JEOD_UNUSED)	src/time.cc	117	1
jeod::JeodBaseTime::must_be_singleton (void)	src/time.cc	137	1
jeod::JeodBaseTime::update (void)	src/time.cc	150	3
jeod::JeodBaseTime::set_time_by_seconds (const double new_seconds)	src/time.cc	178	1
jeod::JeodBaseTime::set_time_by_days (const double new_days)	src/time.cc	194	1
jeod::JeodBaseTime::~JeodBaseTime (void)	src/time.cc	210	2
jeod::JeodBaseTime::add_type_update (const int seeking_status, TimeManagerInit * time_manager_init)	src/time__add_type_update.cc	57	14
jeod::TimeConverter::TimeConverter (void)	src/time_converter.cc	50	1
jeod::TimeConverter::is_initialized (void)	src/time_converter.cc	63	1
jeod::TimeConverter::verify_setup (const JeodBaseTime * master_ptr, const JeodBaseTime * sub_ptr, const int int_dir)	src/time_converter.cc	72	5

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeConverter::can_convert (Direction query)	src/time_converter.cc	120	3
jeod::TimeConverter::convert_a_to_b (void)	src/time_converter.cc	148	1
jeod::TimeConverter::convert_b_to_a (void)	src/time_converter.cc	164	1
jeod::TimeConverter::reset_a_to_b_offset (void)	src/time_converter.cc	180	1
jeod::TimeConverter::verify_table_lookup_ends (void)	src/time_converter.cc	192	1
jeod::TimeConverter::~TimeConverter (void)	src/time_converter.cc	210	1
jeod::TimeConverter_Dyn_TAI::TimeConverter_Dyn_TAI (void)	src/time_converter_dyn_tai.cc	56	1
jeod::TimeConverter_Dyn_TAI::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_dyn_tai.cc	70	8
jeod::TimeConverter_Dyn_TAI::convert_a_to_b (void)	src/time_converter_dyn_tai.cc	161	1
jeod::TimeConverter_Dyn_TAI::~TimeConverter_Dyn_TAI (void)	src/time_converter_dyn_tai.cc	178	1
jeod::TimeConverter_Dyn_TDB::TimeConverter_Dyn_TDB (void)	src/time_converter_dyn_tdb.cc	56	1
jeod::TimeConverter_Dyn_TDB::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_dyn_tdb.cc	70	5
jeod::TimeConverter_Dyn_TDB::convert_a_to_b (void)	src/time_converter_dyn_tdb.cc	135	1
jeod::TimeConverter_Dyn_TDB::~TimeConverter_Dyn_TDB (void)	src/time_converter_dyn_tdb.cc	148	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeConverter_Dyn_UDE::TimeConverter_Dyn_UDE (void)	src/time_converter_dyn_ude.cc	55	1
jeod::TimeConverter_Dyn_UDE::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_dyn_ude.cc	69	5
jeod::TimeConverter_Dyn_UDE::convert_a_to_b (void)	src/time_converter_dyn_ude.cc	136	1
jeod::TimeConverter_Dyn_UDE::reset_a_to_b_offset (void)	src/time_converter_dyn_ude.cc	154	1
jeod::TimeConverter_Dyn_UDE::~TimeConverter_Dyn_UDE (void)	src/time_converter_dyn_ude.cc	165	1
jeod::TimeConverter_STD_UDE::TimeConverter_STD_UDE (void)	src/time_converter_std_ude.cc	55	1
jeod::TimeConverter_STD_UDE::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_std_ude.cc	70	8
jeod::TimeConverter_STD_UDE::convert_a_to_b (void)	src/time_converter_std_ude.cc	143	1
jeod::TimeConverter_STD_UDE::convert_b_to_a (void)	src/time_converter_std_ude.cc	160	1
jeod::TimeConverter_STD_UDE::reset_a_to_b_offset (void)	src/time_converter_std_ude.cc	177	1
jeod::TimeConverter_STD_UDE::~TimeConverter_STD_UDE (void)	src/time_converter_std_ude.cc	191	1
jeod::TimeConverter_TAI_GPS::TimeConverter_TAI_GPS (void)	src/time_converter_tai_gps.cc	56	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeConverter_TAI_GPS::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_tai_gps.cc	70	3
jeod::TimeConverter_TAI_GPS::convert_a_to_b (void)	src/time_converter_tai_gps.cc	109	1
jeod::TimeConverter_TAI_GPS::convert_b_to_a (void)	src/time_converter_tai_gps.cc	121	1
jeod::TimeConverter_TAI_GPS::~TimeConverter_TAI_GPS (void)	src/time_converter_tai_gps.cc	134	1
jeod::TimeConverter_TAI_TDB::TimeConverter_TAI_TDB (void)	src/time_converter_tai_tdb.cc	64	1
jeod::TimeConverter_TAI_TDB::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_tai_tdb.cc	89	3
jeod::TimeConverter_TAI_TDB::set_a_to_b_offset (void)	src/time_converter_tai_tdb.cc	128	1
jeod::TimeConverter_TAI_TDB::convert_a_to_b (void)	src/time_converter_tai_tdb.cc	147	1
jeod::TimeConverter_TAI_TDB::convert_b_to_a (void)	src/time_converter_tai_tdb.cc	163	4
jeod::TimeConverter_TAI_TDB::~TimeConverter_TAI_TDB (void)	src/time_converter_tai_tdb.cc	190	1
jeod::TimeConverter_TAI_TT::TimeConverter_TAI_TT (void)	src/time_converter_tai_tt.cc	55	1
jeod::TimeConverter_TAI_TT::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_tai_tt.cc	69	3
jeod::TimeConverter_TAI_TT::convert_a_to_b (void)	src/time_converter_tai_tt.cc	104	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeConverter_TAI_TT::convert_b_to_a (void)	src/time_converter_tai_tt.cc	117	1
jeod::TimeConverter_TAI_TT::~TimeConverter_TAI_TT (void)	src/time_converter_tai_tt.cc	134	1
jeod::TimeConverter_TAI_UT1::TimeConverter_TAI_UT1 (void)	src/time_converter_tai_ut1.cc	57	1
jeod::TimeConverter_TAI_UT1::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_tai_ut1.cc	85	5
jeod::TimeConverter_TAI_UT1::initialize_tai_to_ut1 (void)	src/time_converter_tai_ut1.cc	135	9
jeod::TimeConverter_TAI_UT1::convert_a_to_b (void)	src/time_converter_tai_ut1.cc	251	7
jeod::TimeConverter_TAI_UT1::convert_b_to_a (void)	src/time_converter_tai_ut1.cc	346	7
jeod::TimeConverter_TAI_UT1::verify_table_lookup_ends (void)	src/time_converter_tai_ut1.cc	441	7
jeod::TimeConverter_TAI_UT1::~TimeConverter_TAI_UT1 (void)	src/time_converter_tai_ut1.cc	489	5
jeod::TimeConverter_TAI_UTC::TimeConverter_TAI_UTC (void)	src/time_converter_tai_utc.cc	58	1
jeod::TimeConverter_TAI_UTC::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_tai_utc.cc	83	6
jeod::TimeConverter_TAI_UTC::initialize_leap_second (void)	src/time_converter_tai_utc.cc	140	9
jeod::TimeConverter_TAI_UTC::convert_a_to_b (void)	src/time_converter_tai_utc.cc	259	8

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeConverter_TAI_UTC::convert_b_to_a (void)	src/time_converter_tai_utc.cc	349	8
jeod::TimeConverter_TAI_UTC::verify_table_lookup_ends (void)	src/time_converter_tai_utc.cc	415	7
jeod::TimeConverter_TAI_UTC::~TimeConverter_TAI_UTC (void)	src/time_converter_tai_utc.cc	459	5
jeod::TimeConverter_UT1_GMST::TimeConverter_UT1_GMST (void)	src/time_converter_ut1_gmst.cc	55	1
jeod::TimeConverter_UT1_GMST::initialize (JeodBaseTime * parent_ptr, JeodBaseTime * child_ptr, const int int_dir)	src/time_converter_ut1_gmst.cc	69	3
jeod::TimeConverter_UT1_GMST::convert_a_to_b (void)	src/time_converter_ut1_gmst.cc	104	1
jeod::TimeConverter_UT1_GMST::~TimeConverter_UT1_GMST (void)	src/time_converter_ut1_gmst.cc	143	1
jeod::TimeDyn::TimeDyn (void)	src/time_dyn.cc	55	1
jeod::TimeDyn::initialize_initializer_time (TimeManagerInit * time_manager_init JEOD_UNUSED)	src/time_dyn.cc	69	2
jeod::TimeDyn::update (void)	src/time_dyn.cc	101	1
jeod::TimeDyn::update_offset (void)	src/time_dyn.cc	119	3
jeod::TimeDyn::~TimeDyn (void)	src/time_dyn.cc	146	1
jeod::TimeGMST::TimeGMST (void)	src/time_gmst.cc	52	1
jeod::TimeGMST::calculate_calendar_values (void)	src/time_gmst.cc	61	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeGMST::set_time_by_trunc_julian (const double nonsense JEOD_UNUSED)	src/time_gmst.cc	78	1
jeod::TimeGMST::~TimeGMST (void)	src/time_gmst.cc	99	1
jeod::TimeGPS::TimeGPS (void)	src/time_gps.cc	52	1
jeod::TimeGPS::set_epoch (void)	src/time_gps.cc	71	1
jeod::TimeGPS::convert_from_calendar (void)	src/time_gps.cc	85	1
jeod::TimeGPS::calculate_calendar_values (void)	src/time_gps.cc	103	1
jeod::TimeGPS::set_time_by_seconds (const double new_seconds)	src/time_gps.cc	120	1
jeod::TimeGPS::set_time_by_days (const double new_days)	src/time_gps.cc	155	1
jeod::TimeGPS::set_time_by_trunc_julian (const double new_tjt)	src/time_gps.cc	171	1
jeod::TimeGPS::~TimeGPS (void)	src/time_gps.cc	190	1
jeod::TimeManager::TimeManager (void)	src/time_manager.cc	62	1
jeod::TimeManager::get_converter_ptr (const int index)	src/time_manager.cc	81	2
jeod::TimeManager::get_jeod_integration_time (void)	src/time_manager.cc	104	1
jeod::TimeManager::get_time_change_flag (void)	src/time_manager.cc	115	1
jeod::TimeManager::get_time_scale_factor (void)	src/time_manager.cc	128	1
jeod::TimeManager::get_timestamp_time (void)	src/time_manager.cc	141	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeManager::get_time_ptr (const std::string& name)	src/time_manager.cc	155	1
jeod::TimeManager::get_time_ptr (const int index)	src/time_manager.cc	170	2
jeod::TimeManager::register_time (JeodBaseTime & time_ref)	src/time_manager.cc	192	2
jeod::TimeManager::register_time_named (JeodBaseTime & time_ref, const std::string& name)	src/time_manager.cc	224	2
jeod::TimeManager::register_converter (TimeConverter & conv_ref, const std::string & name_a, const std::string & name_b)	src/time_manager.cc	253	9
jeod::TimeManager::time_standards_exist (void)	src/time_manager.cc	321	3
jeod::TimeManager::time_lookup (const std::string& name)	src/time_manager.cc	344	5
jeod::TimeManager::update (double current_simtime)	src/time_manager.cc	401	4
jeod::TimeManager::update_time (double current_simtime)	src/time_manager.cc	447	3
jeod::TimeManager::verify_table_lookup_ends (void)	src/time_manager.cc	481	2
jeod::TimeManager::~TimeManager (void)	src/time_manager.cc	502	1
jeod::TimeManager::JEOD_DECLARE_ATTRIBUTES (JeodBaseTime)	src/time_manager__initialize.cc	55	1
jeod::TimeManagerInit::TimeManagerInit (void)	src/time_manager_init.cc	61	1
jeod::TimeManagerInit::initialize_manager (TimeManager * time_mgr)	src/time_manager_init.cc	85	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeManagerInit:: initialize (void)	src/time_manager_init.cc	130	3
jeod::TimeManagerInit:: verify_times_setup (void)	src/time_manager_init.cc	198	9
jeod::TimeManagerInit:: populate_converter_registry (void)	src/time_manager_init.cc	265	10
jeod::TimeManagerInit:: verify_converter_setup (void)	src/time_manager_init.cc	355	7
jeod::TimeManagerInit:: create_init_tree (void)	src/time_manager_init.cc	413	10
jeod::TimeManagerInit:: initialize_time_types (void)	src/time_manager_init.cc	526	3
jeod::TimeManagerInit:: create_update_tree (void)	src/time_manager_init.cc	560	8
jeod::TimeManagerInit:: organize_update_list ()	src/time_manager_init.cc	630	8
jeod::TimeManagerInit::get_ conv_ptr_index (const int index_in)	src/time_manager_init.cc	676	2
jeod::TimeManagerInit::get_ conv_dir_init (const int index)	src/time_manager_init.cc	699	2
jeod::TimeManagerInit::get_ conv_dir_upd (const int index)	src/time_manager_init.cc	725	2
jeod::TimeManagerInit::get_ status (const int index)	src/time_manager_init.cc	752	2
jeod::TimeManagerInit::set_ status (const int index, const int new_status)	src/time_manager_init.cc	773	1
jeod::TimeManagerInit:: increment_status (const int index_slave, const int index_ master)	src/time_manager_init.cc	788	1
jeod::TimeManagerInit::~ TimeManagerInit (void)	src/time_manager_init.cc	806	5

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeMET::TimeMET (void)	src/time_met.cc	63	1
jeod::TimeMET::update (void)	src/time_met.cc	78	4
jeod::TimeMET::~TimeMET (void)	src/time_met.cc	107	1
jeod::TimeStandard::Time Standard (void)	src/time_standard.cc	57	1
jeod::TimeStandard::set_time_ by_seconds (const double new_seconds)	src/time_standard.cc	85	1
jeod::TimeStandard::set_time_ by_days (const double new_ days)	src/time_standard.cc	104	1
jeod::TimeStandard::set_time_ by_trunc_julian (const double new_tjt)	src/time_standard.cc	123	1
jeod::TimeStandard::julian_ date_at_epoch (void)	src/time_standard.cc	142	1
jeod::TimeStandard::add_ type_initialize (const int seeking_status, Time ManagerInit * time_ manager_init)	src/time_standard.cc	157	10
jeod::TimeStandard:: calculate_calendar_values (void)	src/time_standard.cc	283	6
jeod::TimeStandard:: calendar_update (double simtime)	src/time_standard.cc	373	3
jeod::TimeStandard::convert_ from_calendar (void)	src/time_standard.cc	402	1
jeod::TimeStandard:: initialize_initializer_time (TimeManagerInit * time_ manager_init)	src/time_standard.cc	458	24

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeStandard:: initialize_from_parent (Time ManagerInit * time_ manager_init)	src/time_standard.cc	608	8
jeod::TimeStandard::seconds_ of_year (void)	src/time_standard.cc	690	5
jeod::TimeStandard::~Time Standard (void)	src/time_standard.cc	757	1
jeod::TimeTAI::TimeTAI (void)	src/time_tai.cc	47	1
jeod::TimeTAI::set_epoch (void)	src/time_tai.cc	58	1
jeod::TimeTAI::~TimeTAI (void)	src/time_tai.cc	71	1
jeod::TimeTDB::TimeTDB (void)	src/time_tdb.cc	48	1
jeod::TimeTDB::set_epoch (void)	src/time_tdb.cc	59	1
jeod::TimeTDB::~TimeTDB (void)	src/time_tdb.cc	72	1
jeod::TimeTT::TimeTT (void)	src/time_tt.cc	48	1
jeod::TimeTT::set_epoch (void)	src/time_tt.cc	59	1
jeod::TimeTT::~TimeTT (void)	src/time_tt.cc	72	1
jeod::TimeUDE::TimeUDE (void)	src/time_ude.cc	58	1
jeod::TimeUDE::must_be_ singleton (void)	src/time_ude.cc	95	1
jeod::TimeUDE::add_type_ initialize (const int seeking_ status, TimeManagerInit * time_manager_init)	src/time_ude.cc	110	12

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeUDE::convert_epoch_to_update (JeodBaseTime * epoch_ptr, JeodBaseTime * update_from_ptr, TimeManagerInit * time_manager_init)	src/time_ude.cc	238	4
jeod::TimeUDE::initialize_from_parent (TimeManagerInit * time_manager_init)	src/time_ude.cc	307	16
jeod::TimeUDE::initialize_initializer_time (TimeManagerInit * time_manager_init)	src/time_ude.cc	503	23
jeod::TimeUDE::set_epoch_times (JeodBaseTime * epoch_ptr)	src/time_ude.cc	708	4
jeod::TimeUDE::set_epoch_dyn (TimeDyn * epoch_ptr)	src/time_ude.cc	745	20
jeod::TimeUDE::set_epoch_std (TimeStandard * epoch_ptr)	src/time_ude.cc	850	16
jeod::TimeUDE::set_epoch_ude (TimeUDE * epoch_ptr)	src/time_ude.cc	984	13
jeod::TimeUDE::set_initial_times (void)	src/time_ude.cc	1085	24
jeod::TimeUDE::set_time_by_days (const double new_days)	src/time_ude.cc	1198	1
jeod::TimeUDE::set_time_by_seconds (const double new_seconds)	src/time_ude.cc	1212	1
jeod::TimeUDE::set_time_by_clock (void)	src/time_ude.cc	1229	1
jeod::TimeUDE::set_epoch_initializing_value (const double simtime, const double epoch)	src/time_ude.cc	1246	2

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::TimeUDE::clock_update (void)	src/time_ude.cc	1273	4
jeod::TimeUDE::verify_epoch (void)	src/time_ude.cc	1307	21
jeod::TimeUDE::verify_init (void)	src/time_ude.cc	1401	3
jeod::TimeUDE::verify_ update (void)	src/time_ude.cc	1436	4
jeod::TimeUDE::~TimeUDE (void)	src/time_ude.cc	1469	1
jeod::TimeUT1::TimeUT1 (void)	src/time_ut1.cc	48	1
jeod::TimeUT1::set_epoch (void)	src/time_ut1.cc	59	1
jeod::TimeUT1::get_days (void)	src/time_ut1.cc	70	1
jeod::TimeUT1::~TimeUT1 (void)	src/time_ut1.cc	82	1
jeod::TimeUTC::TimeUTC (void)	src/time_utc.cc	48	1
jeod::TimeUTC::set_epoch (void)	src/time_utc.cc	60	1
jeod::TimeUTC::~TimeUTC (void)	src/time_utc.cc	72	1

Bibliography

[1] Generated by doxygen. *Time Model Reference Manual*. NASA, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, July 2023.

[2] Jackson, A., Thebeau, C. JSC Engineering Orbital Dynamics. Technical Report JSC-61777-docs, NASA, Johnson Space Center, Houston, Texas, July 2023.

[3] NASA. NASA Software Engineering Requirements. Technical Report NPR-7150.2, NASA, NASA Headquarters, Washington, D.C., September 2004.

[4] K. P. Seidelmann (editor). *Explanatory Supplement to the Astronomical Almanac, ISBN: 1-891389-45-9*. University Science Books, Sausalito, California, 2006.