Timeline as Unifying Concept for Spacecraft Operations

William K. Reinholtz[1]
California Institute of Technology/Jet Propulsion Laboratory, Pasadena, California, 91109

The notion of Timeline has been used informally in spacecraft operations software for some time, but it has not heretofore been formalized and unified either syntactically or semantically. We have formalized and unified the Timeline so that the commonality can be exploited to reduce the cost of developing and using spacecraft operations software. The Timeline can then be used as the common data structure for storage and communications between spacecraft planning and operations software elements. We have formalized linear timelines (both instantaneous events and functions continuous in time) and activities (potentially overlapping set of activities). Most spacecraft planning and operations processes are naturally expressed in terms of software tools that read timelines from databases as input and generate results as new or modified timelines that are written to the databases. Timelines are rigorously versioned, and each version is immutable, thus a versioned timeline name forever represents exactly the same contents. The name is therefore as good as the contents, and the need for keeping files of contents for communicating between programs, or for associating several timelines or even values on those timelines, or for keeping a record of past values, is eliminated. Timelines thus form the syntactic and semantic method of integration of software elements, leading to decreased adaptation cost. Operations efficiency is increased because historically segregated elements are easily integrated so that there are fewer gaps in the operations process that must currently be closed, if they are closed at all, by expensive or inefficient means.

I. Introduction

THe AMMOS Mission Operations System tools and services are being reengineered in order to improve its capability and efficiency, and reduce operations and maintenance costs[1]. Timelines were identified as a ubiquitous data structure within the existing AMMOS, though they were noted to be implicit, informal, and non-uniform in representation and semantics, and so they could convey little practical benefit to the current AMMOS[2]. Timelines were subsequently proposed as a key architectural concept and concrete data structure for the new AMMOS[3], and were in the final stages of vetting as of April 2012.

The proposed AMMOS architecture is based on the notion of the orchestrated execution of software (programs, processes, services) with the bulk of the data being of Timeline semantics and syntax, and data exchange being primarily through the Timeline database. It is an orchestrated, blackboard architectural style, with Timelines as the foundational semantics and consequent data structure. The semantics are specified a priori and are an architectural invariant.

There are several major programmatic and technical advantages to this approach:

- It decouples the programs from each other so that they can be adapted and evolved independently. (AMMOS principle: Minimize coupling). Maintenance and Adaptation costs are then reduced.
- It makes the program interactions explicit so that it can be managed and evolved in a systematic manner. Maintenance costs are then reduced.
- Communications, Coordination, Information Model, and software are separable so that each can be managed independently. For example, moving from software to services is facilitated. Communications can be changed (e.g. between message bus and RESTful) without rewriting the software. Exploiting the tremendous scalability of the cloud is straightforward.
- Incidental coupling due to ad-hoc point-to-point interfaces and data structures is avoided.

[1] JPL Principal Engineer, MS 301-480, 4800 Oak Grove Drive, Pasadena, CA 91109-8099, Email: kirk.reinholtz@jpl.nasa.gov

- It makes it relatively easy to add new programs to the ensemble, which reduces the temptation to construct large monolithic applications. Adaptation and Maintenance costs then reduced.
- It allows smaller, more cohesive programs to be constructed and integrated (AMMOS principle: Maximize cohesiveness). Adaptation and Maintenance costs are then reduced.
- It facilitates effective program interactions across organizational boundaries because of common language of Timelines. Comprehensive closed-loop control of the spacecraft at various levels, Integrated Planning and Sequencing, comprehensive Accountability: All are made practical.
- Programs can be orchestrated in new ways, to provide new capabilities.

The following list outlines some of the many spacecraft operations that are organized around timelines in AMMOS. All of these operations manipulate timelines, though the timelines are at present tacitly defined and the definitions and have not heretofore been formalized (or even expressed, for that matter) or shared amongst the capabilities. That lack of sharing has lead to stove-piping that has hindered cost-effective interoperability.

- Record telemetry channel values as received
- Compute spacecraft state timelines from telemetry
- Creating command timelines from plan/activity timelines
- Updating expected spacecraft state timelines based on commands to be sent to the spacecraft
- Comparing expected spacecraft state with actual spacecraft state so as to adjust future plans and activities
- Graphical and textual real-time display of timeline values to monitor spacecraft health and performance
- Graphical and textual display of timeline historical values to monitor spacecraft health and performance
- Automated flight rule checking
- Science planning
- Spacecraft engineering activities
- Subsystem monitoring and trending and health analysis
- Science product availability prediction and notification

This paper defines the key timeline concepts (immutability, versioning, timeline type, timeline name, timeline instance, and timeline value), key types (state, measurement, event, and activity), and several mission operations software architectures that exploit timelines to reduce costs and improve spacecraft operability.

II. Related Work

The work described here is one of several tasks related to the modernization of processes and software used within the NASA-funded Advanced Multi-Mission Operations Systems (AMMOS) program. The Operations Revitalization task (OPSR)[1] is developing an AMMOS architectural framework and set of Mission Services that include adaptable multi-mission engineering specifications. These specifications include essential documentation such as operational concepts and scenarios, requirements, interfaces and agreements, information models, and mission operations processes. Those products will provide clear and rigorous specification of operations that inform the work described here.

The second task, called Sequence Revitalization (SEQR) and the subject of this paper, is developing detailed timeline semantics and data structure specifications, and most of the software architecture and implementation that uses timelines.

Both tasks use timelines as a key architectural concept, but treat them from somewhat different perspectives. The Operations Revitalization task is concerned with the usage of timeline information by operations personnel - the human need for information. The SEQR task focuses on the detailed structure of information and how it can be manipulated using software to service human needs for information. This has led to different levels of detail in the two tasks' respective timeline information models and implementation. The timeline information models are joined into a unified and coherent whole in [2] as part of the Ops Revitalization task.

It was recognized that realization of the full potential of these related tasks required that they work to a common set of architectural principles and objectives, as otherwise there would doubtless be clashing assumptions and implementation, along with semantic gaps and conflicts, that would be costly to resolve (and would probably damage the integrity of the architecture until resolved). The several teams therefore participated in the development of a common architectural vision, the results of which are described in [3].

III. Introduction to Timelines

The timeline[4] is abstractly defined as a container of items indexed by time, or of items related by time. The abstract definition is intentionally rather open, and the edge between timeline and not-timeline is fuzzy. The

abstract definition does not need to be formalized, because it's the types of timelines that are actually defined that have practical impact. Timelines are made practical by creating concrete types of timelines that can be precisely defined, stored in databases, manipulated in software, and so on. An informal example follows to illustrate the concept of timeline. The definition of new broad types of timelines is a matter of detailed system engineering and engineering judgment and is not further discussed here.

- Record Measurements – Many telemetry channels are measurements, e.g. a periodic sampling of the spacecraft bus voltage. Such measurements are recorded on timelines (usually one timeline per sensor). The timeline is a conceptually a list of timestamp and value tuples, where timestamps must have values and must be unique within a given timeline. The timestamp is defined to be the primary key of the tuple. In this case the timestamp is the time at which the sensor was read, and the value contains the sensor reading.
- Record State - "… state variables are the smallest possible subset of system variables that can represent the entire state of the system at any given time. ("State space (controls)," *Wikipedia, The Free Encyclopedia*). Measurements are sampled in discrete time. State in principle has a value at all points in time. A state timeline typically implemented as piecewise continuous interpolators. We require that all state timelines return a value for any possible time $0..\infty$, as otherwise individual software elements would hard-code the meaning of values (probably conflicting with other elements and probably not logically reasonable for all timelines) outside the domain of the timeline. Measurement and State timelines may well share common structure (tuple indexed by time yielding sample, or tuple indexed by time yielding interpolator). We have found it best to architect around their mathematical distinctions, rather than implementation similarities.
- Record Events - A Labeled instant in time. Events have zero duration. "The switch turned on" for example, or entries in a time stamped error log. ("Event (Relativity)," *Wikipedia, The Free Encyclopedia*).
- Record Commands – A timeline that represents the information received (or may be received) by the spacecraft (or more generally, the system being controlled, e.g. a DSN antenna) for the purpose of changing its behavior.
- Record Intent – Intent timelines (often expressed today as activities or plans) indicate an acceptable envelope of spacecraft operation: Intent is not a single state trajectory. It is, at least abstractly, a set of possibilities. The general act of operating a spacecraft uses intent to determine specific commands to impose on the spacecraft, and reconciles the resulting state with the intent in order to determine what adjustments to intent must be made and what further commands must be uplinked. It is basically closed loop control between intent and state.

IV. Key Timeline Concepts

A **Timeline** is informally and most generally defined as a mathematical construct that contains items that are strongly temporally related (e.g. the estimated battery voltage as a function of time, or the discrete finite list of bus voltage measurements indexed by time of sample acquisition, or "this and that must not both happen at the same time" or "this must happen no more than 5 minutes after that") We prefer to describe timelines in mathematical terms, rather than those of software engineering, to keep focus on the engineering use of the timeline (e.g. a function continuous in time that represents the battery voltage at any point in time) rather than on the implementation details of the timeline (e.g. a series of linear interpolators). We in particular do not want mere implementation details to leak into all the software that uses the timelines, which would make it very costly to change those details (basically they aren't mere details once that leak happens).

Several mathematical timeline types may use the same data structures (e.g. a list of events indexed by time and a list of interpolators indexed by time may use the same data structure, but are quite distinct mathematical concepts). Such sharing is simply an implementation detail and must not leak past the API, because we must to be able to evolve the data structure without performing costly changes to all the software that uses the timeline.

Not everything that relates to time is a timeline. If time is the dominant index and/or relationship in the structure, it's probably a timeline. If not, it may not be a timeline. For example, you could store all science images received from the spacecraft on a simple timeline that's indexed by time and yields the image that was acquired at that time (we call such a timeline an event timeline). However, that's probably not the dominant access pattern and there are many other likely query forms and relationships, so this probably isn't the most appropriate use of a timeline.

Best practices suggest that you use the simplest timeline type that will work for your application. For example, you can do everything with a temporal constraint network timeline that you can do with an event timeline, so in theory the event timeline is unnecessary. But practical considerations (performance, simplicity, robustness, …) bring great utility to the linear timeline: *"In theory there is no difference between theory and practice. In practice, there is"*

Various implementation mechanisms are used to map an instance of a mathematical/system-engineering timeline into a software data structure for storage and transmission. The architecture is flexible on this point so that the mapping can be selected based upon the use cases for the particular timeline. For example, an estimate of the spacecraft bus voltage is a mathematically a function continuous in time, and may be mapped into software as piecewise continuous polynomial interpolators.

The values of a timeline are stored in a **timeline instance** (typically but not necessarily in a database of some sort). Each timeline instance contains multiple **timeline versions**. In principle every mutation of the timeline instance may form a distinct version, though in practice the programmer may choose to perform a number of mutations as an atomic operation. The timeline instance is in effect an L-value [cite Wikipedia value] that has a name, and provides a name for each mutation (e.g. write or assignment) to the instance. A reference to every mutation of any instance can thus be created and dereferenced in a uniform manner, providing referential transparency.

Every timeline instance is assigned a unique Timeline ID number ("**TLID**") by the system when the instance is created. Two timelines are defined as the same timeline if they have the same TLID, and otherwise they are not the same timeline (identical copies perhaps, but not the same timeline). Renaming a timeline is defined as associating the new name with the ID that the old name previously referenced. Copying a timeline is defined as creating a new ID with the new name, and performing in effect a deep copy (for performance reasons one may choose to be clever in sharing structure, though the effect will always be that of a deep copy) of data into the new ID.

Every timeline instance is given a unique **timeline name.** Names are **namespaced**: For example, each mission would probably have its own namespace, so that timeline names need only be managed at the project level. Namespaces basically work like Linux directories: There is a "root", the root has directories, directories have directories, and directories may contain timeline names. The timelines within a namespace need not reside in the same database and may be moved amongst databases without changing the name: migration and tiered storage is thus supported. Immutability means that they may also be mirrored for replication purposes. Names are assigned to timelines by the users, and may be changed. The architecture *per se* does not depend on the structure of the namespaces, but software and processes no doubt will, and so it will be a (very important) matter of system engineering to design an appropriate namespace scheme. The scope of the namespace in effect defines the scope of the system boundary: If two independent namespaces are created, then the same name may refer to two different timelines, which means that those two namespaces must be understood to define non-overlapping system boundaries.

A version of a timeline instance, once created, is at once and forever **immutable**. It follows that the timeline instance full name and timeline version, taken together, will always reference exactly the same value, assuming that the indicated version exists. The story is a bit more complicated, but still sound, when renaming is allowed. The story is also a bit more complicated, but still sound, when physical deletion is allowed. More on these complications later.

When a reference to a particular version of a timeline instance is dereferenced, the result is a **Timeline Value**. A timeline value is an R-value. Most of the AMMOS software will probably manipulate either versioned instance references, or values. Explicit manipulation of the versions will likely be the domain of configuration management.

Immutability of timeline versions means that timeline values are referentially transparent: The value can always be recreated from a reference to the version, and so no timeline value can exist that isn't a version of a timeline instance. It is never necessary to store a file containing a timeline value, because the value can always be recreated from the versioned timeline name. This is the primary reason for specifying the immutability principle at the architectural level.

Timeline versioning (in fact all versioning, including mutable timeline metadata and even non-timeline information) is modeled in terms of the **SCN** ("System Change Number"). The SCN is in effect an integer that is incremented by at least 1 within each transaction: The SCN must be allocated in strictly increasing order (though they may not be exposed in the same order, due to transaction semantics); holes are allowed so that certain performance optimizations may be applied; and duplicates are never allowed. The SCN is used to label each change made within each transaction, where transactions are in effect serialized (we only literally serialize them where semantics depend on serialization, and otherwise reserve the option for non-serialized execution for performance reasons). The SCN represents the instant in time at which the resulting database state is defined to have been created or logically deleted. The SCN therefore precisely labels each state of the database, and no state exists which is not labeled by an SCN. Each record is in effect tagged with the SCN at which it was created, and the SCN at which it was logically deleted. The record is defined as logically existing for all SCN: SCNCreated <= SCN < SCNdeleted. The creation SCN must not be changed after initial creation of the record. The deletion SCN is given the value of infinity at record creation, and can be mutated exactly once thereafter to assign a deletion SCN. An

implementation may choose to log each SCN along with pertinent details as to who did it, why, when, etc, to provide a detailed change log of each change to the state of the DB.

Physical deletion is allowed. Any operation that references a deleted SCN will return an appropriate error indication. An error must be indicated, because a deletion would otherwise cause violations of immutability and so referential transparency. The architecture depends upon immutability, and physical deletion is a practical necessity, and so an error is thrown. Physical deletion is only allowed when it is proven that the deletion can't alter the results of any query (other than to throw the error noted above), for that would violate immutability. For example, if there were a "how many records in the database as of SCN x"? query, then any physical deletion would alter that query for all time into the future. Such queries must not be provided for other than administrative purposes, as otherwise physical deletion would be prohibited.

Transactions are only required to be serialized where the system depends upon the serialization: for example, mutations to a given timeline instance would generally be serialized so that a long-lived transaction can not cause violations of immutability, but mutations to distinct instances probably don't need to be serialized. System engineering determines what serialization models should be used.

Every timeline instance has **timeline metadata** that describes the static (i.e. can not be mutated once the instance is created) and dynamic (can be changed over the life of the instance) properties of the timeline. Mutable metadata (e.g. the timeline name) is subject to SCN semantics. The metadata is used make explicit the information that is needed to find and use the timeline (contrasted with imbedding the information in the tools that use the timeline). For example, the schema (or "type") of the items in the timeline should be in the metadata, so a timeline visualization tool could in principle read the schema and display the timeline using only that schema.

Since the timeline instance name may be changed and is therefore under SCN semantics, access via name is a two step process: (1) The name is dereferenced to a TLID with respect to a given SCN; then (2) the TLID is dereferenced to data in potentially another SCN. We expect that in the common case, the same SCN will be used for both operations, though the architecture supports the use of distinct SCN's.

Every item in a timeline instance must have a key that uniquely identifies that item within version(s) of the instance: Often the index is a time value, but may be more complicated. It follows that name, version, and item name forms a durable and immutable **timeline item reference**. That reference forms the basis of data structures that track relationships between items. For timeline structures where there is as well-formed notion of the location of, or index of, an item in the timeline, that location or index should serve as the item name: no point in making up new names when existing ones will do. For example, the time index of an event timeline is a perfectly good name for the item at that time index.

Every timeline instance is immutably associated with the time system (e.g. TAI) of the time values within the instance, and the time format (e.g. large integer fixed point offset from epoch in nanoseconds) in which those values are stored within the instance. The time system must include the location of the clock, so there is no confusion when relativistic effects are significant. For example, an atomic clock onboard a particular spacecraft, even a perfect one, forms its own time system. Various mechanisms for conversion between commensurate time systems, and presentation via display formats, are of course provided.

V. Timeline Categories

The major types, (or classes, or categories), of timelines is an open, but highly controlled, list. It is open because we expect that as new technologies and techniques come along, we will need to extend the list of supported timeline types. It is highly controlled to counter the apparently natural human tendency to define a new type with semantics very specific to the application at hand, rather than use a previously defined type. If the list is too easily extended, then, type proliferation will dilute the key advantage of having common timeline semantics standardized and shared amongst many applications.

Timeline categories have two distinct perspectives: The system engineering perspective, which is primarily concerned with the semantics of and operations upon the various types; and the implementation types, which encode the semantics into computer data structures that can be serialized and versioned and stored.

A. System Engineering Categories

- Measurement – A measurement timeline contains sampled measurements, usually obtained from a sensor of some sort, that are recorded as a totally ordered time-indexed sequence of data points. There is typically one timeline per sensor. Multiple data points at the same time index are not allowed, because of the confusion that typically follows when various tacit, implicit, and/or ad-hoc mechanisms are inevitably used to force an order onto the duplicates. If multiple points can occur at the same time

instant then a composite data point type is defined that can contain multiple samples, and any required ordering semantics if required are made explicit as part of the definition of that composite type. Measurements are generally input to estimation processes, which result in state timelines. Processes other than estimation should generally operate on state timelines, rather than raw measurement timelines.

- Event – An event is a labeled instant in time. Event timelines are often used to label points in time at which events did or are intended to occur. For example, an event timeline might be used to mark the instant in time at which each telemetry pass may first expect to receive a signal. As with measurement timelines, the events are totally ordered and time indexed. Events are defined in this way to prevent their creep (via the addition of durations, or allowing them to consume resources or directly cause things to occur) into the semantic domain of state or intent timelines.
- Command - A timeline that represents the information that has been or may be by the spacecraft (or more generally, the system being controlled, e.g. a DSN antenna) for the purpose of changing its behavior.
- State – The engineering concept of system state (for example the state of the spacecraft) is that the state at some point of time is sufficient to predict how the system will respond to inputs (gravitational forces, commands, etc) without reference to past history of the system inputs and outputs. "The State" is obviously a very large and generally unknowable value. Much of the art of system engineering is determining a useful subset of that platonic state that can be known (often estimated from measurements) and is required to operate the spacecraft. That subset is known as the state variables, where one variable may be the attitude of the spacecraft, another may be the velocity, yet another the acceleration, another for camera articulator angles, available fuel, battery state of charge, commands queued up for execution, etc etc etc. The spacecraft state is represented by state timelines (probably tens of thousands of them), usually one timeline per state variable. There are also state variables for other systems of interest to the spacecraft operator, for example rover surface assets, the configuration of the Deep Space Network antennas, and telemetry relay assets. A state timeline is abstractly a function of time that yields a state variable value at that time. The key distinction between a measurement timeline and a state timeline is that the domain of the measurement timeline is a finite list of timepoints, and the domain of a state timeline is the infinite set of all possible timepoints. The other distinctions noted here are a matter of sound engineering practice but not intrinsic to the architecture. If you interpolate between measurements, you're using the measurement timeline as a state timeline and really should use a state timeline instead. State timelines are total in time (yield a value for any possible time input, from birth of universe to infinity beyond), to avoid the mess that would come if applications themselves encoded the meaning of times not covered by the state timeline. It is better for the timeline to return an explicit "spacecraft did not exist at that time" than to encode that interpretation into all software that uses state timelines.
- Intent - Spacecraft operations often involves creating activities that represent higher level desired spacecraft behavior, which are then decomposed into commands that can be executed by the spacecraft and that accomplish the activities. Telemetry is monitored, perhaps by comparing actual state timelines (computed in turn from measurement timelines) to predicted state timelines (perhaps generated via simulation), so as to assess and manage progress in executing the activity. An activity timeline contains activity instances. An activity instance is a named and parameterized interval of time, where the start and end time of the interval may be assigned values, or may be variables or allowed ranges of times. The actual semantics of the activity name and parameters is outside of the scope of the timeline itself. We expect that several intent timeline types will be developed to support spacecraft operations automated planning and scheduling system. A survey of such systems and the possibilities of developing common timeline semantics for them is covered in [5].

B. Implementation Types

The system engineering timeline categories are implemented as shown here. There is not a one to one relationship between the system engineering category and the implementation mechanism, because in some cases a single mechanism can support several implementation types without compromise.

Every timeline instance ("timeline instance" as defined herein and elaborated upon in [4]) must have the following associated information (not just the slots: values must be assigned):

- Timeline type - Immutable.

- TLID – Immutable, assigned by system at time of creation.
- Timeline name – Mutable, may be changed after timeline created.
- Time system – Immutable.
- Time format – Immutable.
- Physical Schema – Immutable.
- Type-specific information as specified for the timeline type

Every timeline version must be associated the information necessary to allow the receiving entity to interpret the bits, and must contain the information necessary to retrieve the bits from the timeline instance that contained the version. It is not necessary to serialize this information with each version: it's OK to use the timeline name or TLID as a primary key ("PK") for this information where that makes sense, for example.

- All information specified as common to every timeline instance
- The query that extracted the information from the instance
- The instance SCN to which the query was applied

Every timeline instance is assigned a name (where the name consists of a namespace prefix followed by a short name). The name uses path syntax and semantics ("Path," *Wikipedia, The Free Encyclopedia*). The name does not define the location of the timeline (e.g. it is not a URL to a particular database), though an important property of the syntax is that it's designed to be used as part of an HTTP URL without the need for character encoding should one wish to construct a URL from the name. Specifically, the names are constructed with reference to RFC 2616 and RFC 2396.

The name syntax is intentionally rather restricted. This is so that the name is not likely to collide with symbols used in other protocols and languages.

The timeline name has the following syntax as specified in Augmented BNF as defined in RFC 2616:

```
TLNAME = *1 ( "/" ALPHA * (ALPHA | DIGIT | "_"|"-") )
```

1. *Grounded Linear Timeline ("GLTL")*

The GLTL is used to store system engineering timelines of type Measurement, Event, and State.

The GLTL is the simplest timeline type. It is purposefully restricted to a simple form for two reasons: (1) A large number and percentage of timelines used in spacecraft operations are of this simple type; and (2) We want a simple, theoretically sound, easily explained, easily implemented timeline for the ubiquitous case. Where more complexity is required, different timeline types are used.

A grounded linear timeline is defined as a sequence of {timepoint, item} tuples where:

- All timepoints are represented in the same time system and in the same time format; and
- All timepoints are grounded, meaning they are assigned specific absolute values (not variables, not offsets relative to previous timepoints); and
- The timepoints have a total order, meaning in particular that there are no duplicate values.

State Timeline discretized values (e.g. "ON " or "OFF") are stored directly, with the common semantic that the value holds over the interval from the timepoint of that item in the timeline, to but not including the timepoint of the following item.

State Timeline Interpolated values may be stored in two ways:

- Store discrete values in the timeline, then compute and evaluate the interpolation function as needed. In this case the interpolation function is computed and discarded every time it is used. The NAIF Type 8 ephermis ("http://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/spk.html#Supported%20Data%20Types ") is an example of this approach.
- Store the interpolation function in the timeline, where each function is defined as valid over the interval from the timepoint of that item in the timeline, to but not including the timepoint of the following item. The NAIF Type 2 ephermis is an example of this approach.

The purpose of this specification is to define the minimal syntax, semantics, and attributes of grounded linear timelines such that they may be exchanged between systems without semantic clashes or ambiguity, and with a minimum of syntactic conversion machinery. We expect that concrete bindings will be developed (e.g. XML schemas and documented CSV formats) but those bindings are not part of this specification.

The purpose of a Grounded Linear Timeline ("GLTL") is to store and communicate things that happen at an instant in time (an "event") and things that have values that vary with time. The definition is very broad, as is the application of the GLTL. An event might be a telemetry frame and the time might be the ERT of the first bit of the frame. It might be an EVR, indexed by SCLK of the EVR. The notion of an "event" is that it is most naturally considered to "happen at a time". The other category of GLTL information stored in a GLTL is that which varies with time: The voltage on a spacecraft bus, the position of the sun relative to the spacecraft at a given time, the

number of people on the JPL campus at any time. The notion here is that the information is most naturally considered as having a value at all times.

For time varying values the GLTL tuple value for a given timestamp is applicable from that timestamp, to but not including the following timestamp. There is no separate "duration" value: that is strictly implied by the timestamps. If you find yourself wanting to describe things that have overlapping durations, you're probably into a non-linear timeline representation or you are looking at the problem the wrong way. You could force it into the linear timeline with ad-hoc information in the TLVALUE field, but that's probably an indication that the GLTL is the wrong representation for whatever you're trying to describe, or that you defined your timelines incorrectly. As an example of the latter point, consider the mechanical pointing of all DSN antennas. If you try to put it all on a single timeline, well, that's wrong and leads to the overlapping durations problem. If you have a timeline for each antenna that records where it is mechanically pointing at any time, that's a classic GLTL representation.

A GLTL is defined as a list of tuples <TIMESTAMP, TLVALUE> where:

1. TIMESTAMP is in the same time system and time format (as defined by NAIF) for each tuple in the GLTL; and
2. The time system and time format is specified; and
3. TIMESTAMP is assigned a literal value (“grounded”); and
4. The list is totally ordered by TIMESTAMP (no duplicate values of TIMESTAMP); and
5. The “type” of TLVALUE, which is herein defined as its structure in mathematical terms of integers, reals, enumerations, sets, lists, … is specified; and
6. The “physical schema” of TLVALUE, where is herein defined as the representation of the mathematical structure in terms of serialized computer constructs such as ints, floats, lists, … is specified; and
7. The “interpretation” of TLVALUE, which is herein defined as the semantics of the “type” (e.g. that the list of floats is interpreted as coefficients of an interpolation function) is specified; and
8. If the GLTL represents a value over an interval of time, the interval is defined as starting from TIMESTAMP and continuing to but not including the following TIMESTAMP (there must never be a separate “duration” concept); and
9. If the GLTL represents a value over an interval of time, the first interval in the timeline must start at TIMESTAMP=0 and the last interval is defined as covering TIMESTAMP..Infinity; and
10. If the GLTL represents an event at an instant in time, the time at which the event occurred is defined as TIMESTAMP (there must never be a separate “occurred at” concept).

Some design notes:

- “Type” and “Interpretation” are distinct concepts so that we can reuse a type (e.g. list of floats) via different interpretations (polynomial coefficients, Chebyshev coefficients).
- “Physical Schema” and “Type” are distinct concepts so that the same Type (e.g. list of floats) may be serialized into different physical schemas (XDR, ASN.1, Protocol Buffer, Thrift) as needed. The physical schema can then be evolved to suit size/performance requirements at hand without altering the software that uses the timeline.
- The interpretation should include units if the notion makes sense for the GLTL at hand. There should be a standard for expressing interpretations so that things that are common to many (but not all, for otherwise we would specify it outside of the interpretation) GLTL types (such as units) can be extracted by software in a standard manner.

2. *Activity*

An activity timeline is a type of intent timeline that is in common use in the current AMMOS system.

An activity instance has:

- activity type
- activity instance id (unique, never recycled)
- parameter values
- start timepoint
- end timepoint

where a timepoint is:

- minimum time
- maximum time
- expected time (between min and max)

Time slots must all have assigned values: Use 0..Infinity to indicate wide open values

VI. Architecture Principles

A number of principles, outlined here, guided development of our architecture. Our objective in developing and maintaining this list was to describe the "culture" of our architectural work, so that the architecture might not rot as team members come and go over the life of the project.

1. **A Domain-Specific architecture is good** – The purpose of AMMOS is to operate spacecraft. The architecture does not need to extend to other domains (for example, AMMOS won't be used to implement a banking system). This principle allows us to avoid over-generalization of the architecture by making commitments as to what the architecture will not do, which leads to lower adaptation costs by reducing the number of decisions that must be made for every adaptation and reducing the volume of infrequently used concepts and code.
2. **Timelines are the central organizing concept and data structure** – Timelines are ubiquitous in spacecraft operations, as described earlier in this paper as well as [1] and [2]. A key factor of our architecture is that the semantics of timelines are specified in detail a priori. Any data structure that is reasonably directly mapped to those semantics can then be converted to another representation, which makes the details of the data structure something of an implementation detail since easy conversion is a priori known to be straightforward. That said, sound engineering suggests that data structures should be reused where reasonable so as to avoid the expense of writing and maintaining low value conversion code.
3. **Immutability - Strive towards immutable data structures** – Immutability means that once a particular version of a particular data structure is created, that version is never further mutated. A reference to that version thus forms a reliable reference to the bits in the version. In the case of timelines, the architecture itself then supports queries of the general form "As of last Friday noon, what was the predicted bus voltage midnight Saturday?". Tracking data provenance becomes practical, because every version of every data item has a well-formed reference that can be used to track the many important data relationships found in spacecraft operations systems.
    - Immutability applies across the system: information about or relating timelines must follow the same principle in order for the architecture as a whole to provide immutability
    - Eliminates vast range of potential bugs, most of which are basically pointer errors writ large
    - Eliminates much ad-hoc code that provides localized immutability
    - Provides first-class names for every version of every data structure instance
    - Forms a solid basis for modern stateless internet protocols such as REST ("Representational State Transfer," *Wikipedia, The Free Encyclopedia*)
4. **Good theoretical foundation to the architecture** - A sound implementation unlikely to result of unsound architecture!
5. **Minimally constraining architecture - But the rules that are there, are hard and fast** – Architectural mandates become the foundation of the implemented system, so changing the mandates is naturally very expensive. We strive to mandate only things that are essential to the integrity of the architecture. So, for example, the architecture does not require a particular language (Java, say) or a particular communications style (pub/sub for example). Those are important design decisions, of course, but they are not architectural. We strive to constrain only where the "physics" of the situation makes such constraints a good long-term bet.
6. **Data structure, not data stream –** Spacecraft generally emit a telemetry stream, which tends to lead to systems that are architected around that data stream: The stream is parsed, processed, displayed, and archived. Focus on the steam in turn leads to a tendency to leave access to historical data as an implementation detail. Each endpoint on the telemetry stream must deal with archiving the data for historical access, recovering historical information on startup, and so on. Our focus is on the overall timeline, with uniform access to the past and present.
7. **Exploit but isolate vendor-specific capabilities** – We want to exploit best-of-breed technologies (e.g. replication services in high performance databases), so that we don't have to build such capabilities ourselves on a less powerful foundation. However, we also must be prepared to replace the product with another that doesn't have the same capabilities. We meet these seemingly conflicting needs by using the vendor capabilities, but not allowing the vendor-specific details to "leak" across the system as a whole. We strive to move such advanced capabilities into "'ilities", rather than operational necessity, so there's a good trade between cost and capability. Then if an advanced feature is needed (say realtime transactional replication), we can either use a product that implements it (an expensive commercial database product) or implement the advanced feature ourselves on top of a "free" database. As another example, we use SQL databases, but the SQL is not available across the system: Queries are supported via a neutral interface, and the query is executed behind that interface in whatever terms the underlying database technology requires.

8. **Generally prefer solutions with a minimum of "machinery"**
9. **Evolvable and Scalable**
10. **Security levels "adjustable" and the architecture does not limit how far you turn the knob either way**

VII. Architecture

The AMMOS System structure will have 6 major elements, described below. This is true no matter how the system is designed (For example, changing database technologies, or using web-based tools, does not change this structure). The system is structured in this manner so that the system elements have interfaces that are more a consequence of the architecture than the design, so that the design and implementation of each element can evolve without disrupting the overall architecture.

- Timeline semantics and data structure – Much of the value of the AMMOS architecture will come from the unified manner in which the many time-oriented data structures used in spacecraft operations are represented. The Timeline is that unifying data structure.
- Timeline Database – The architecture is based on the execution of coordinated processes that operate on timelines that are stored in one or more Timeline Databases ("TLDB"). The database in general stores the timelines such that time-related aspects are reflected in the DB schema or structure, but the values related to the times are opaque to the database. The database structure thus forms a stable foundation for the system, because it does not need to be changed as new timeline types are introduced. In general the TLDB can only perform time-oriented lookups. That said, it is plausible that if extreme TLDBDB performance is necessary and if that performance depends upon value-based operations (not just time based) then the TLDB will need to exploit the structure of the timeline values themselves. This will not break the architecture but it will probably increase adaptation and maintenance costs.
- Timeline Database Library – There is executable code in front of the database and executing on the server side (assuming a server side) that converts higher-level technology-neutral queries into DB-specific primitives, and also processes the results. For example, the server code might perform a bulk query, to minimize the number of DB interactions. The client code can't perform such operations itself, because the DB product or technology choice would then leak into the client code and make it difficult to change DB products or technologies. The server side may also support plug-ins to present project-specific views of some timelines in a uniform manner. That could be done on the client side, but may be less costly and easier to maintain if done once and for all on the server side.
- Client library – The client library element is a library that is compiled with the application program (e.g. compiled with APGEN) that provides an application-oriented interface to the timelines. The architecture constrains this library much less than the other elements, so that the library can rapidly evolve to contain functions that are found to be useful to the application programmer. The client library contains the bulk of the code that requires knowledge of the semantics of a particular timeline type or instance. There may be multiple libraries, each for a different purpose, e.g. analytics vs realtime vs modeling.
- Orchestrator – The Orchestrator coordinates the execution of the various programs that are run to perform mission operations. A project may choose to use more or less orchestration. The orchestrator may interact with workflow if a project uses formal workflow mechanisms. Program execution can also be initiated manually, or triggered by timeline database events. The orchestrator may be explicit (e.g. a BEPL engine), or may be encoded into pipe-and-filter, dataflow, whatever.
- Name Server – The name server returns the current physical location(s) of a timeline, given the name of the timeline. It's function is analogous to the internet DNS. Note that the scope of the nameserver effectively defines the boundary of the system, because two independent nameservers allow the same name to refer to two different timelines and therefore must be independent, distinct systems.

A. TLDB

The Database Interface is a key architectural invariant, along with the Timeline. It is designed to allow the database technology to be selected to meet mission needs. For example, a small mission may choose to use a free database. A larger mission may choose to use a commercial database that provides robust hot backups, offsite mirroring, local caching, cloud scaling to infinitely, and the like. A mission may choose to put the data in a commercial Cloud. It may even change DB technologies over the life of the mission, using something cheap and light in formulation, heavy in operations, and optimized for archival access in perpetuity. The interface is stays the

same no matter what technology is used by a project, so that the spacecraft operations software suite will operate the same regardless of the DB technology used.

The point is that the database technology itself (SQL, NOSQL cloud, whatever) is not an architectural invariant or "load-bearing wall", and so can be changed as needed as long as it meets the interface specifications. The interface is architectural, and much of the up front engineering rigor in the definition of timelines and the operations they support is there so that the interface will not need to be modified later. "Interface" as used here does not mean the details of the communication fabric (RESTful, message bus, etc). Those must remain design choices. "Interface" means the basic protocols: what functions are performed, the data types that go in and out, that sort of thing. It's been done right when it's easy to switch from one communication style to another. It's been done wrong if doing so becomes a big deal. Don't let communication fabric details leak into (or even become) the architecture.

The TLDB interface provides fairly direct access to the DB data structures. Client-side libraries perform further conversions into application-specific forms. A thin server-side library is between the interface and the database and converts the raw timeline information in the database into the forms that are used by the client software (and vise versa). The idea is that the server-side library is an "adaptor" between the fixed API behavior on one side, and the implementation specifics of the chosen database technology on the other side. It thus isolates the database technology from the client software, so that the database technology can be selected based on its 'ilities (e.g. performance, cost, comprehensiveness, …) without having to do a costly and risky rewrite of the client software. Basically, it avoids a lock into the DB technology. In contrast, if we exposed SQL or JDBC at the interface level that would lock into at least relational databases, and quite likely would lock into a particular vendor. Timelines were designed such that such a commitment is not necessary.

The Server-side interfaces will be the minimal set that provide a computationally complete set of operations on the timelines, so that the interface is likely to prove stable and is likely to support all desired abstractions as they evolve. "Code comes and goes, Data is forever" – The timeline schema is the most strongly vetted, followed by the database interface, then the official library code, and finally the abstracted services. Orderly evolution is thus supported, with room for ad-hoc extensions at the client level and an evolutionary path for appropriate extensions to be made available for more general use or even inclusion into the architecture itself.

B. Libraries

The Client-side Library computes the runtime and application-specific presentation of the timelines. For example, the database representation of a given timeline (say state timeline on the battery state of charge) might be stored a time-tagged list of interpolators with parameters encoded into XDR and sampled at a fixed interval, say every hour. The Client-side Library would provide several views of that timeline. It would provide a floating-point value of the battery SOC for any time over $0..,\infty$ and yielding either a voltage or other information, for example that the spacecraft did not exist at the selected time. It may also provide a more detailed, type-specific interface that gives access to detailed uncertainty information and the like.

The client library provides mathematical operations that can be performed on timelines, for example differencing and integration. Such operations are provided in a functional programming style, so that the caller (typically application-specific code in a component) need not express incidental machinery (e.g. a for loop) in order to perform computations on timeline. That isolation allows the library to operate in several modes (bulk computation, real-time as new data arrives, lazy evaluation) without modification of the user code.

We expect that several variations on the library will exist, each presenting a different "personality": Perhaps one for extremely high performance and fairly direction interaction with the TLDB; and another that presents a mathematically oriented interface for analytics and simulation(as described above).

C. Components

A large number of the computations that are performed by the current AMMOS can be easily reinterpreted as one of several types of computations performed on timelines, and yielding timelines. For example, there are many derivations performed, such as computing power as the product of voltage and amperage, and many checks performed such as checking that the power does not exceed some limit (where the limit is either a constant, or itself a timeline). Other computations, such as simulation-based predictions, are more complicated. In any case, such computations are performed by components that have several important properties:

- The component is fairly isolated from its execution environment, so it can be repurposed (e.g. computing the power product as a one-shot computation, or performing it in real-time as new voltage and amperage values become available).
- The component generally does one thing (check that a timeline does not exceed a value), rather than many things (check it does not exceed a value, send out a textie alert to subscribers if that limit is

exceeded) so as to maximize reusability. The components are then composed into higher-level capabilities (e.g. compose the limit check component that generates a Boolean timeline with another that sends a text if the Boolean timeline ever becomes True)

D. TMS

The Timeline Management System ("TMS") is the name for the system comprised of one or more TLDB's, one or more libraries, many components, the orchestration mechanism for executing the components in order to do mission operations, and a few other elements like the timeline name registry. The TMS is a web highly scalable and cloud-compatible server with a RESTful API, so that it's easy to manipulate via modern web-aware scripting and programming languages.

E. Clients

A client is defined as any system outside of the TMS that is using the TMS to perform mission operations. For example, a sophisticated simulator may read its initial state from the TMS timelines and write its computed state timelines back into the TMS. That simulator would be defined as a client since its execution is not orchestrated from within the TMS.

F. Name Server

The programmatic scope in which names are managed in a single rooted namespace (in other words, the level at which uniqueness is guaranteed) is an important decision. It is likely that the scope will be much larger than a single database instance, and so there will be a name server that covers many instances. Within that scope, names and TLID's and SCN's are known to be unique so the timelines will play well together no matter what database they are in. The nameserver is very much like the internet DNS ("Domain Name System," *Wikipedia, The Free Encyclopedia*) in that it associates various information (e.g. the TLID and location) with the timeline name. The location can thus be changed without recoding the users of the relocated timeline.

VIII. CM Operations

Many processes within the MOS involve the application of CM processes on timelines.[2]. The architecture, via the semantics of the SCN and Namespaces, can implement a number of CM models in a simple and straightforward manner:

First write wins/Optimistic Locking – The SCN of the version to be edited is recorded. Upon write, the recorded SCN is compared to the SCN of last mutation. The write succeeds only if they match. If the write fails, the caller must in effect redo the complete read/edit/write operation, based on the current version of the timeline. This model basically mimics what happens when two people edit the same file, and the editor says "file on disk has been modified".

Last write wins – The SCN of the version to be edited is recorded. Upon write, any logical insertions performed since the recorded SCN are logically deleted, and any logical deletions performed since the recorded SCN are re-inserted. The write is then performed. The write can't fail in this model. This model basically mimics what happens if two people edit the same file: the last person to write the file completely obliterates whatever the other people wrote.

Pessimistic lock – The timeline is "locked" before editing is started, and the lock is only released after the edits have been written to the timeline. Other editors that attempt to acquire the lock during that time will fail to acquire the lock. This model mimics various source code control systems (SCCS, RCS, SVN…) that provide for locking a file to force serialized edits.

Branch/merge – A set of timelines, or segments thereof, are deep copied from the parent timelines into new timelines for editing. The SCN of each parent timeline is recorded at the time of the copy. The user can read/write the new timelines without concern for conflicts, because the user is the only one that can read or write those copied timelines. When the user is done with the edits, any changes applied to the parent timelines since the deep copy are merged into the workspace, and then the merges are posted back into the parent timelines. If any parent was updated since the merge, the write fails. The user repeats the merge/write cycle until the write succeeds. The merge does not need to preserve the total semantic correctness of the parent: errors will be caught later during consistency checks. The object of the merge is to minimize the odds of errors being found, but it need not guarantee there are no errors. If the full set of consistency checks are quickly and easily performed, they may be applied at the time of the merge in order to greatly reduce the odds of a later error, but again, that is only an optimization, not a requirement.

IX. Administrative Operations

1) Physical deletion of certain SCNs – The immutability principle in its pure form implies that no data can ever be physically deleted, because the SCN in which the data did exist may be queried at any time in the future. However, physical deletion is a practical necessity in order to manage disk space. The immutability principle is thus extended to require that any possible query against any past SCN always returns the same result (this is the "pure" clause), or, it returns an error indicating that the query is no longer possible do to physical deletion (this is the extension to the definition). Physically deleting SCNa..SCNb is defined to mean that any query outside of that range will function normally, and any query within that range will return an error. Physical deletion is implemented thus:
   a) SCNa..SCNb are entered into a "Deleted SCN's" table.
   b) All queries raise an error if the queried SCN is in the Deleted SCNs table.
   c) A extended SCN range is constructed that is the given range plus any contiguous previously deleted ranges either side of the given range
   d) Any row that was both created and deleted within the extended range is physically deleted

   It should be noted that administrative database operations are by their nature not immutable.
2) Physical deletion of timeline instance

   All rows related to the given timeline are physically deleted using normal DB deletion mechanisms, except that the TLID is marked as physically deleted rather than physically deleted from the TLID table. This marking is not strictly necessary, but does provide more robust error messages and could be the basis of automatic forwarding of requests to the proper database.
3) Moving a timeline

   The new location of the database is entered into the Nameserver. Note that this type of move is administrative and so by definition not subject to SCN semantics.
4) Splitting a database

   Create the new database, then use "move a timeline" administrative operation to populate the new instance and "physical deletion of timeline instance" to remove it from the source database.
5) Merging databases

   All rows in the source database are copied into the destination database. This "simply works" because the TLID, SCN, and name are global across all instances. Physical deletion is then performed on the source database.

X. Future Work

Much of the theory and implementation proposed here can be applied to repositories in general, not just timelines. The generalization is fairly straightforward: Consider timelines a type of repository, add an association to each TLID (which should be renamed Repository ID ("REPID") indicating the type of repository it references), and use the immutability and SCN mechanisms to store other repository-type information.

For example, we are exploring the development of a file repository, where the REPID indexes a row containing among other things a binary column of essentially unlimited size in which information usually stored in literal files can be placed. The existing SCN and immutability mechanisms would provide robust and rigorous versioning and naming and security and audit logging etc far beyond what a file system can provide, for only a few days of additional effort.

Acknowledgments

K. Reinholtz thanks the many people who devoted time and intellect to the constructive critique of these concepts in various presentations and reviews over the last three years. This work is greatly improved thanks to your contributions.

References

[1]Bindschadler, D.L., Delp, C.L., Principles to Products: Toward Realizing MOS 2.0, SpaceOps 2012 Conference, Stockholm, Sweden, Jun 11-15, 2012 (to be published),

[2]Chung, S.E., Timeline-based Mission Operations Architecture, SpaceOps 2012 Conference, Stockholm, Sweden, Jun 11-15, 2012(to be published),

[3]Estefan, J.,AMMOS Architecture Vision, NASA/JPL MGSS DOC-000780, 2012,

[4]Reinholtz, W.K., Timeline Central Concepts, JPL Technical Report D-71055, 2011,

[5]Chien, SA., Knight, R., Johnston, M.,Cesta, A., Frank, J., Giuliano, M., Kevalaars, A., Lenzen, C., Policella, N., Verfaille, G., A generalized timeline representation, services, and interface for automating space mission operations, SpaceOps 2012 Conference, Stockholm, Sweden, Jun 11-15, 2012(to be published),