

Timeline and the Timeline Exchange Infrastructure: A Framework for Exchanging Temporal Information

Kenneth Donahue Seung H. Chung

Jet Propulsion Laboratory, California Institute of Technology
Pasadena, CA 91109
626-590-8340

{Kenneth.M.Donahue, Seung.H.Chung}@jpl.nasa.gov

Abstract—The concept of a *timeline* is used ubiquitously during space mission design and development to specify elements of flight and ground system designs; it is used also during testing and operations to describe mission plans and system state trajectories, for example. In this paper we introduce our *Timeline Ontology*. The Timeline Ontology is grounded in mathematical formalism, thus providing concrete semantics. We also describe our ontology-based *Timeline eXchange Infrastructure (TXI)*, a framework that provides a means to exchange time-varying information among various tools and algorithms with semantic correctness. To further ground the needs for Timeline and the TXI, we examine the tools used in the Mission Operations Systems (MOS) at the Jet Propulsion Laboratory (JPL). To illustrate the versatility of the formalism, we also describe a use of Timelines during the early design phase of a project lifecycle. Finally, we look at future extensions to this work, including creating a user interface for Timeline instance editing, integrating with ontology validation tools, and extending the Timeline concept to include relationships to other pre-existing JPL ontologies.

itly such that exchanging data between tools becomes very challenging. Finally, with a lack of semantics, computing with time-varying information (e.g. computing the time that specific events occur and therefore the value of a specific variable at a specific time) becomes non-trivial.

The Need for a Timeline Ontology

In this paper we present our *Timeline Ontology*. The Timeline Ontology is grounded in mathematical formalism, thus providing concrete semantics. Because of its mathematical basis, computing using Timelines is natural given appropriate mathematical tools and algorithms. Timeline instances can be manipulated (i.e. variables, constraints, and events can be added, removed, or mutated) by various analysis tools, e.g. schedule planners and schedule executors. As such, a Timeline instance can be used to store various types of temporal information, e.g. the input for a schedule planner, the output schedule of a planner, the input for a schedule executor, a trace of an as-executed schedule, etc. For this paper, we will be focusing on using a Timeline instance to represent a trace of an as-executed schedule, but want to emphasize that this is only one possible use.

TABLE OF CONTENTS

1	INTRODUCTION	1
2	RELATED WORK.....	2
3	TIMELINE.....	2
4	TIMELINE EXCHANGE INFRASTRUCTURE (TXI)	4
5	TXI TOOLS	5
6	APPLICATIONS.....	6
7	NEXT STEPS	8
	REFERENCES	9
	BIOGRAPHY	9

1. INTRODUCTION

The concept of a *timeline* is used ubiquitously during space mission design and development to specify elements of flight and ground system designs; it is used also during testing and operations to describe mission plans and system state trajectories. For example, mission scenarios and plans are commonly referred to as timelines; additionally, the time history of telemetry, commands, and system states are commonly visualized as timelines. In current practice however, the notion of Timeline is only used conceptually. Instead of using one unified Timeline representation, information is represented in various forms. Furthermore, while the information representations may have a well-defined syntax, their semantics are usually non-existent or defined only implic-

Timeline eXchange Infrastructure (TXI)

This paper also describes our ontology-based *Timeline eXchange Infrastructure (TXI)*, a software framework that provides a means to exchange time-varying information among various tools and algorithms, as depicted in Figure 1. This approach allows us to exchange time-varying information with semantic correctness.

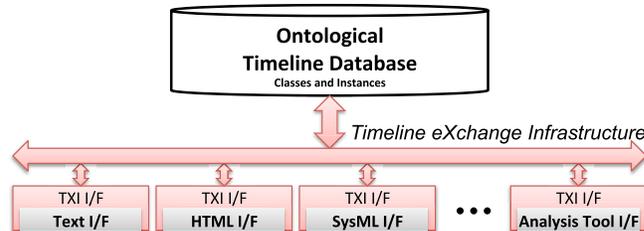


Figure 1. A simplified depiction of the Timeline eXchange Infrastructure (TXI).

Our preliminary TXI implementation allows many different visualization, analysis and editing tools to communicate with one another. This implementation includes 1) a domain-specific language (DSL) for representing a Timeline instance, 2) a Resource Description Framework standard (RDF/XML) representation of a Timeline instance, and 3) a centralized database for storing and retrieving the *Timeline definition*

978-1-4577-0557-1/12/\$26.00 ©2012 IEEE.
¹ IEEEAC Paper #2270, Version 1, Updated 02/01/2013.

(the Classes specified in the Timeline Ontology) as well as many *Timeline instances*. Using TXI, we are able to store Timeline instances and transform them between various formats, allowing the data to be visualized in various forms (e.g. diagrams, mathematical equations, tables, plots) that are useful for better understanding the dynamic aspects of a system design.

Using the TXI as a base, we were able to additionally create Timeline-related tools. First, we created a Timeline Solver for computing values of Timeline instance variables, including instant event times as well as dependent variables over time. Timeline instant event times are specified as t_n , and are related to other times via constraints, such as $t_n = t_{n-1} + 5.0$, ultimately linking back to the start time of the Timeline. The Timeline Solver computes numerical values for each of those variables; using that information, we can then solve for the dependent variable values over time. Second, we built upon the Timeline Solver, creating a Timeline Viewer for plotting the dependent variable of a Timeline instance over time.

Timelines in System Design and Mission Operations

To further ground the needs for Timeline and the TXI, we examine the tools used in the Mission Operations Systems (MOS) at the Jet Propulsion Laboratory (JPL). For a given project, analysis tools (e.g. power analysis tools, telecom analysis tools, attitude analysis tools, sequencing tools) are integrated as necessary in a rather *ad hoc* manner. These integrated tools exchange data in various formats, where the exchange can only occur with customized conversion scripts. Due to lack of well-specified semantics for these formats, developing these conversion scripts is non-trivial and the effort required for their development is highly uncertain. Accordingly, the project has to allocate extra time and effort to develop these scripts. By using Timeline as the common representation between these tools, this adaptation overhead can be mitigated, reducing the resources required by the project to perform this tool integration.

We also describe a use of Timelines during the early design phase of a project lifecycle. Specifically, we found Timeline to be useful for storing and analyzing scenario-based power load profiles and power and energy margins. While discussing this use case, we will focus on the ability to compute using Timelines.

Finally, we look at future extensions to this work, including creating a user interface for Timeline instance editing, integrating with ontology validation tools, and extending Timeline to include relationships to other pre-existing JPL ontologies.

2. RELATED WORK

Attempts to model variables changing over time has been done in various ways.

The Architecture Analysis and Design Language (AADL) allows the modeling of synchronous data flows within a system. The Unified Modeling Language (UML) and Systems Modeling Language (SysML) provide Timing Diagrams, Interaction Diagrams, and Sequence Diagrams. These diagrams have notions of TimeConstraints which can be used to express durations between events. UML MARTE is a UML extension that focuses on modeling real-time embedded systems.

While all of these options have many features that are

Timeline-like, there are a few features missing: 1) Constraints on variables (ValueProperties) are handled in an ad hoc manner and 2) although these diagrams are good for showing specific execution traces, they are less good at showing the entire space of possible behaviors for a system.

We strove to create a system that could accomplish all of these things.

Previous work has been done in defining time ontologies with the intent of performing analyses based on temporal logic [1][2]. The current incarnation of the described *Timeline Ontology* does not extend from these efforts, but we are looking into which ontologies would be beneficial to import.

3. TIMELINE

A Timeline constrains time-varying information over time. A simple and common example of time-varying information is a position $x(t)$ of some object. While the codomain of a time-varying position is a three dimensional vector, i.e. $D(x) \in \mathbb{R}^3$, a timeline may be associated with more complex and less conventional codomains, e.g. a set of discrete data, such as camera images. In the case of camera images, the dependent variable would be the image state and events would represent image processing operations, e.g. gaussian blurring, channel extraction, thresholding, hough transformations, etc.

For a more formal mathematical definition of Timeline, please refer to [3].

Timeline Ontology

Generally speaking, we define *Timeline* as a representation of time-varying information; specifically, a representation of *events* in time. While some only associate the word “event” with an occurrence at a specific point in time, we use a more general and broader definition of *event* that includes a notion of an occurrence over a period of time (a time interval). To disambiguate these usages, we denote an event at a given time as an *instant event* and an event over a time period as a *durative event*.

To create a rigorous definition of Timeline, we used Protégé [4], an open-source ontology editor which can export ontologies written in the Web Ontology Language (OWL) [5] to the Resource Description Framework (RDF) standard, a World Wide Web Consortium (W3C) specification [6]. Any standards-based ontology editor would be fine; we chose to use Protégé because of its user interface, export to various standard representations, and its Java interface. In this section we first describe how OWL is used to define terminology and then define the Timeline ontology.

OWL: Class and NamedIndividual

The Web Ontology Language (OWL) standard has a distinction between *Classes* and *NamedIndividuals*, where a *NamedIndividual* is an individual or “instance” of a *Class*. This terminology originates from philosophy, and subsequently Artificial Intelligence, under which a class is a *category of being* or concept and an individual is an instance that is a member of a class. It is possible to specify both *Classes* and *NamedIndividuals* in a same database. We chose to define our *Classes* in a separate database from that of the *NamedIndividuals*.

The organization of our databases is depicted in Figure 2. The

first database represents the database that stores the Timeline ontology, i.e. it stores the definitions of the terminology associated with Timeline. The second database stores Classes of Events (including Classes of Timelines) that are specific to certain domains. The third database stores the NamedIndividuals of Timelines and Events that are domain-specific. While each database is depicted as a single database, each database may be partitioned further and/or distributed. As is depicted in the figure, we currently store the domain-specific Classes with the NamedIndividuals. We may decide to break the Classes into a separate database as the system scales.

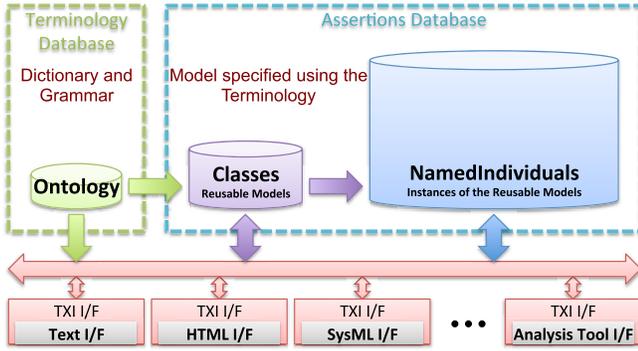


Figure 2. A depiction of the Timeline eXchange Infrastructure (TXI) and its ontological database types.

This organization scheme provides a separation of concerns. The Timeline Ontology database is expected to be a concise set of terminology and relationships that is expected to remain static over time as it is independent of domain. The Class database is expected to be domain-specific, but reusable through instantiation. The NamedIndividual database is expected to be non-reusable and domain-specific.

Although the Timeline Ontology database is small, the rationale for storing it in a database is that we envision this Timeline Ontology connecting into other existing ontologies. We believe all our ontologies for an entire project should live in a centralized database, which allows all the ontologies to be analyzed with formal ontology reasoning tools. For example, although the Timeline Ontology currently stands alone, we envision tying it into existing ontologies which define Components, Interfaces, Flows, etc.

Timeline Classes

The Timeline Ontology was developed based on work by Chung [3]. In this section, we describe the Timeline Ontology, but refer the readers to [3] for formal mathematical definitions of Timeline.

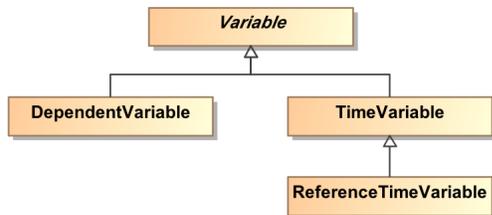


Figure 3. Diagram showing the inheritance of different Variables. (The white-tipped arrow is the generalization/specialization relationship indicating the ‘is a’ relationship).

Variables—There are three types of *Variables* in the Timeline Ontology. We plan to add additional constructs for *Constants* and *MathFunction*, but for this iteration, defining the following *Variables* was sufficient.

- *TemporalVariables* - A Variable that represents an instant in time. These are used to bound the temporal extent of events. They can be constrained relative to one another to create durations or intervals;
- *ReferenceTimeVariables* - A special TemporalVariable that represents time t , not just an instant in time. There should only be one of these per Timeline.
- *DependentVariables* - A Variable that is dependent on the *ReferenceTimeVariable*. A Timeline (in our current framework incarnation) has a reference to a single DependentVariable; all other DependentVariables in the Timeline’s contained Events are assumed to be related to the Timeline’s DependentVariable via some mathematical operation. For example, a Timeline could represent current best estimate (CBE) for power for a Flight System, so $powerCBE[FlightSystem]$ would be the Timeline’s DependentVariable, and part of that could be the CBE Power for a Payload, so $powerCBE[Payload]$ would be another DependentVariable that relates back to $powerCBE[FlightSystem]$ as part of a summation function.

The relationships between Terms are shown in Figure 3.

Events—The basic building block of a Timeline is an Event. An Event is meant to represent a period of time (or instant in time) where certain Constraints are imposed on the Timeline. Events come in a few distinct flavors, but the main discerning characteristic is whether or not an event has any temporal extent (has a duration) or occurs instantly (does not have a duration). The different Event types described below are depicted in Figure 4.

- *InstantEvent* - An Event that does not have any temporal extent. It occurs at a specific moment in time, so it has a reference to a particular TemporalVariable;
- *DurativeEvent* - An Event that does have temporal extent. As such, it persists from one InstantEvent (the start) to another InstantEvent (the end). A DurativeEvent can additionally contain other Events. It is related to the start InstantEvent by the “hasStart” relationship, to the end InstantEvent by the “hasEnd” relationship, and to its contained Events by the “containsEvent” relationship;
- *Timeline* - A special DurativeEvent that has a relationships to the root DependentVariable via “hasDependentVariable” and to the ReferenceTimeVariable via “hasReferenceTimeVariable”; and
- *CompositeTimeline* - A special Timeline where all immediately contained Events are Timelines themselves. The “containsEvent” in this case has been specialized and redefined to be “containsTimeline” and only point to Timelines.

Constraints—A Constraint is an element which restricts the value of a Variable. A Constraint is related to Variables by a “couples” relationship and related to Events by a “constrains” relationship. There are two types of Constraints defined thus far. They are described below and depicted in Figure 5.

- *TemporalConstraint* - A special Constraint that links multiple TimeVariables together. For example,

$$e1_{end} == e1_{start} + 5$$

would be a duration constraint that would force the duration of $e1$ to be 5 time units; and

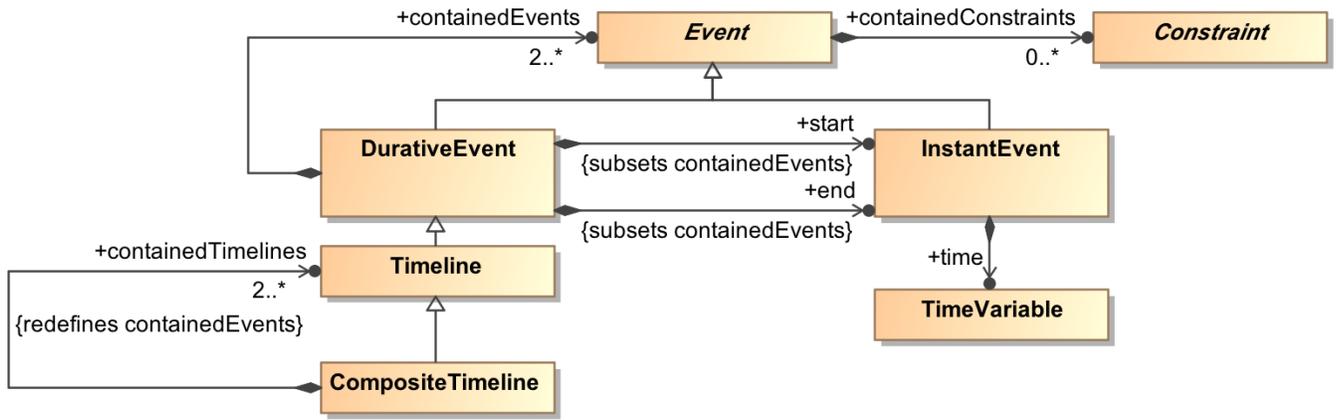


Figure 4. Diagram showing the inheritance of different Events as well as some relationships between Events. (The black-diamond is a Directed Composition relationship and indicates the ‘has a’ relationship).

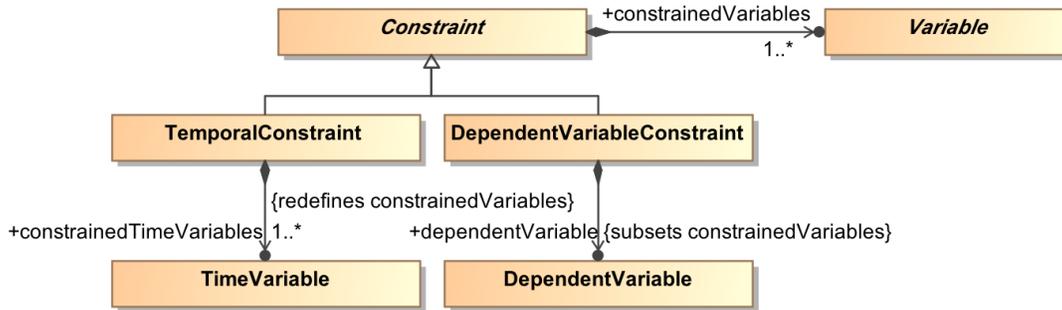


Figure 5. Diagram showing the inheritance of different Constraints some relationships between Constraints and Variables

- *DependentVariableConstraint* - A relationship between multiple *DependentVariables* and/or a *ReferenceTimeVariable*. For example,

$$powerCBE[FlightSystem] == powerCBE[Payload] + powerCBE[Bus]$$

or

$$powerCBE[foo] == 1.5 \cdot \ln(t - e1_{start}).$$

Additional types of constraints will be explored further in future iterations of this infrastructure.

Example NamedIndividual

NamedIndividuals were used to store Timeline instances. An snippet from a specific Timeline instance is shown in Figure 6.

4. TIMELINE EXCHANGE INFRASTRUCTURE (TXI)

The Timeline eXchange Infrastructure (TXI) was created to support the exchange of temporal information between various formats and tools. It is a centralized repository for temporal information along with utilities for querying that information in a meaningful way.

Timeline Database

To store the Timeline Ontology and to store Timeline instances, we used Sesame [7], an open-source framework

for storing RDF data in a queryable database. This type of database is commonly called a “triple-store” as the RDF data comes in triples: <Subject, Predicate, Object>. Our plan is to extract the reusable behaviors into Event Classes to store in the Classes database, but this has not yet been implemented.

Java-based TXI API

We decided to use Java as the language for our TXI glue because many tools come with the ability to add in Java plugins. Additionally, our database (Sesame), ontology editor (Protégé), and various tools (e.g. Maple, Mathematica, MagicDraw) all have Java interfaces.

We are looking into using Alibaba, an open-source tool in Sesame which can be used to generate Java code from an ontology. This would mean that the Timeline portion of the TXI API would always be consistent with the defined Timeline Ontology. As it currently stands, the two are independent entities and consistency between the two is managed manually.

Standard Timeline Transformations

The TXI API includes the classes and structure for Timeline, but in order for that to be useful, the API additionally needed the ability to transform Timeline information between standard formats. While the Java-based Timeline objects are useful at runtime, they are not helpful for storing information for offline use or for later retrieval. For those uses, it is better to store a Timeline instances in a database or file.

The format exchanges we provide are:

```

<!--
////////////////////////////////////
//
// Individuals
//
////////////////////////////////////
-->
<owl:NamedIndividual rdf:about="&LukeDemoLaunchScenario;LukeDemoLaunchScenario">
  <rdf:type rdf:resource="&timeline;Timeline"/>
  <dc:description rdf:datatype="&xsd:string">null</dc:description>
  <timeline:containsEvent rdf:resource="&LukeDemoLaunchScenario;LukeDemoLaunchScenario.event22"/>
  <timeline:containsEvent rdf:resource="&LukeDemoLaunchScenario;LukeDemoLaunchScenario.event13"/>
  <timeline:containsEvent rdf:resource="&LukeDemoLaunchScenario;LukeDemoLaunchScenario.event16"/>
  <timeline:containsEvent rdf:resource="&LukeDemoLaunchScenario;LukeDemoLaunchScenario.event11"/>

```

Figure 6. This is a screenshot of a NamedIndividual file representing a Timeline instance. Specifically, this NamedIndividual stores the power information for the SMAP’s Launch Scenario. See Section 6 for more information.

- Timeline File Parser - Parses a Timeline instance RDF/XML file or Timeline instance DSL file and generates a corresponding Timeline object in Java.
- Timeline File Writer - Takes a Timeline object in Java and writes a Timeline instance RDF/XML or DSL file.
- Sesame Querier - Queries Sesame for Timeline instance information given a timeline name and generates a corresponding Timeline object in Java.
- Sesame Publisher - Takes a Timeline object in Java and pushes all RDF statements in it to the triplestore database.

All other transformations must be written for the specific application.

Expected Usage

We expect the Timeline API to be included in a plugin to a specific tool. The plugin would use the Timeline API to generate, mutate, and destroy Timeline instance information in various formats.

As an example, we created a plugin to MagicDraw which included the Timeline API. The purpose of the plugin was to 1) transform a SysML Activity to a Timeline instance; 2) save the result in the Sesame database; and 3) write an RDF/XML file locally. This tool has been successfully implemented and tested. It is discussed in detail in Section 5.

5. TXI TOOLS

In this section we will discuss some of the tools we have created for performing operations on Timeline instances.

SysML Activity To Timeline Instance Transform

SysML is used on several tasks at JPL for modeling various facets of systems including behaviors, structure, requirements, test procedures, test configurations, etc. In order to integrate the ideas of Timeline into those tasks, we decided to add in an Activity-to-Timeline-Instance transformation capability.

SysML has many of the concepts we need for defining Timeline concepts; where it fell short, we were able to augment SysML by creating stereotypes to capture the additional data and metadata. We only found the need to create a single additional stereotype: << Scenario >>.

The << Scenario >> stereotype allows the user to specify certain fields there are relevant for Timeline. Specifically, the stereotype has a field for the collection of Dependent-Variables of interest, a field for the root component of the physical decomposition (which is used to determine “rollup” constraints), and several string fields for capturing the variable names used for t_{start} , t_{end} , $t - t_{start}$, and t .

The mapping from a SysML Activity to a Timeline instance is as follows:

- SysML Constraints constraining an Activity or an Activity Node are mapped to TemporalConstraints.
- Actions (specifically CallBehaviorActions) are mapped to DurativeEvents. As DurativeEvents can contain other Events, we follow the Action’s Behavior back to an Activity to find its child-Actions.
- Initial and Final Nodes are mapped to Start and End InstantEvents of a DurativeEvent.
- Fork and Join nodes are mapped to InstantEvents.
- ActivityFlows (Control Flow and Data Flow) are mapped to TemporalConstraints that replicate the token-flow of how Activities execute. That is to say that an ActivityFlow that indicates that an Action starts when the previous Action ends is mapped to a Temporal constraint equating the start time of a DurativeEvent to the end time of the previous DurativeEvent.
- All other aspects of SysML Activity Diagrams are unsupported.

Timeline Solver

As a proof of concept, we wanted a tool that could take a perfectly constrained Timeline instance and solve for all the unknown variables.

For example, imagine a simple Timeline instance T for $x(t)$ with two serial DurativeEvents, A and B . Each DurativeEvent has a start and end InstantEvent with a corresponding TimeVariable, T_{start} , T_{end} , A_{start} , A_{end} , B_{start} , and B_{end} . Let us further assume that:

- T starts at $t = 0$;
- A lasts for 5.0 time units and restricts x such that $x == 1.0$; and
- B lasts for 7.0 time units and restricts x such that $x = 0.0$.

These variables are all related to one another as follows:

- $T_{start} == A_{start}$, from structure;
- $A_{end} == B_{start}$, from structure;
- $B_{end} == T_{end}$, from structure;
- $T_{start} == 0.0$, from given start time;
- $A_{end} == A_{start} + 5.0$, from given duration;
- $B_{end} == B_{start} + 7.0$, from given duration;
- and the dependent variable constraint

$$x(t) == \begin{cases} 1.0, & A_{start} < t < A_{end} \\ 0.0, & B_{start} < t < B_{end} \\ \text{undef}, & \text{otherwise} \end{cases}$$

This is a perfectly specified Timeline instance. Each variable can be unambiguously determined from this information. We first solve for each of the TimeVariables, and from that, assuming we have the piecewise definition of the dependent variable, we can determine the dependent variable at any point in time.

As in our example, the Timeline Solver starts by extracting all the constraints from the Timeline instance. All temporal constraints and dependent variable constraints are written to a file in solver-specific syntax. The purpose of this file is twofold. First, it provides a trace that allows us to debug any issues with the transformation. Second, it allows for a snapshot of the Timeline instance that can be used in the solver for various analyses (instead of having to run the transformation each time). We chose to use Maplesoft’s Maple [8] for our solver, but the transformation could just as easily have written source code for another symbolic solver tool.

The solver is asked to solve for the values for all the TimeVariables so that the temporal extent of each part of each piecewise equation is known. Once we had a “solved” piecewise representation of each DependentVariable over time, we can query the solver for the DependentVariable we care about at any level by specifying both the DependentVariable and the time at which we want to know its value.

Timeline Viewer

The Timeline Viewer was built on top of the Timeline Solver. It uses the Timeline Solver to solve a perfectly constrained Timeline instance. It then samples a specified DependentVariable in that solved Timeline instance n times from $t = T_{start}$ to $t = T_{end}$ and plots the results on a graph in a new window. The time-sampled data is also written to a file for external analyses if needed.

6. APPLICATIONS

While working on this project, we focused on two distinct stakeholders who need to store data in the form of timelines. These stakeholders were 1) Flight System Engineers, who require information in the form of timelines for various scenario-based analyses; and 2) The Mission Operations System and Ground Data System (MOS/GDS) engineers, who require temporal information of resources in order to determine schedules for various activities.

Application for Flight System Engineering (SMAP)

The Soil Moisture Active/Passive (SMAP) mission is a flight project with a joint radiometer/radar instrument which will be used to map soil moisture and the freeze-thaw cycle of water on the Earth’s surface. We took notional scenarios from SMAP and put them in our Timeline format so we could

look at the power usage of each component, subsystem, and system over time.

Modeling Power for SMAP’s Launch Scenario—The SMAP project chose to model flight system power consumption over various scenarios using a mixture of artifacts, including

- An Activity-like diagram representing the notional sequence of events along with the components responsible for performing those events (see Figure 7);
- A Sequence spreadsheet representing the actual flight system configuration, specifying the power mode for each component for each event; and
- A Mode spreadsheet enumerating the modes of each component along with a power consumption static value for each of those modes.

While SMAP’s approach to modeling sequences works, there is one major shortcoming. There is no direct connection between their Activity-like Diagram (the graphical representation of the sequence) and the Sequence spreadsheet (the sequence play-by-play). This means that both documents need to be updated manually to ensure that they are synchronized. By using a Model-Based Systems Engineering approach to this modeling effort, the disconnect between the graphical and analytical models can be remedied.

Using Timeline for Power Scenario Modeling—We began by modeling the components in SysML, showing how those components are collected into subsystems and ultimately systems. This physical decomposition imposes “rollup” constraints in our Timeline instance. That is to say that the DependentVariable (e.g. *powerCBE*) of a component in our physical decomposition tree is related by some mathematical expression (e.g. RSS, RMS, or in this case, a summation Σ) to that component’s children’s DependentVariables. These constraints are not added explicitly in the SysML model as constraints; they exist implicitly based on the physical decomposition. As part of the transformation from SysML Scenario to Timeline instance, these implicit constraints are added explicitly to the Timeline instance. There is no analogy to this in SMAP’s modeling approach as structural information is implicit based on the equations in the spreadsheet and general layout of the spreadsheet. Because these physical-decomposition-based constraints are independent of the Scenario, future work will include adding these constraints to the database for all Timeline instances to reference, instead of creating similar constraints for each Timeline instance, which is what we currently do.

Once we had a physical decomposition of our system, we created a behavior specification for each of the leaf-level components, specifying their operational modes and transitions between operational modes. This specification is analogous to SMAP’s Mode spreadsheet. However, unlike the Mode spreadsheet, which presumes constant power consumption on a mode, these modes can have dependent variable constraints (in this case power consumptions) that are arbitrary functions of time, using references to both absolute time (t) and relative time (time since entry into that mode, $t - t_{start}$). Another distinction is that our specification has an understanding of the mode transitions, making certain sequences of modes impossible.

Upon completion of this behavior specification for each of the components, we generated Scenarios (see Figure 8). There are no rules on the structure of the Scenario’s Activity tree. One assumption made during the transformation from

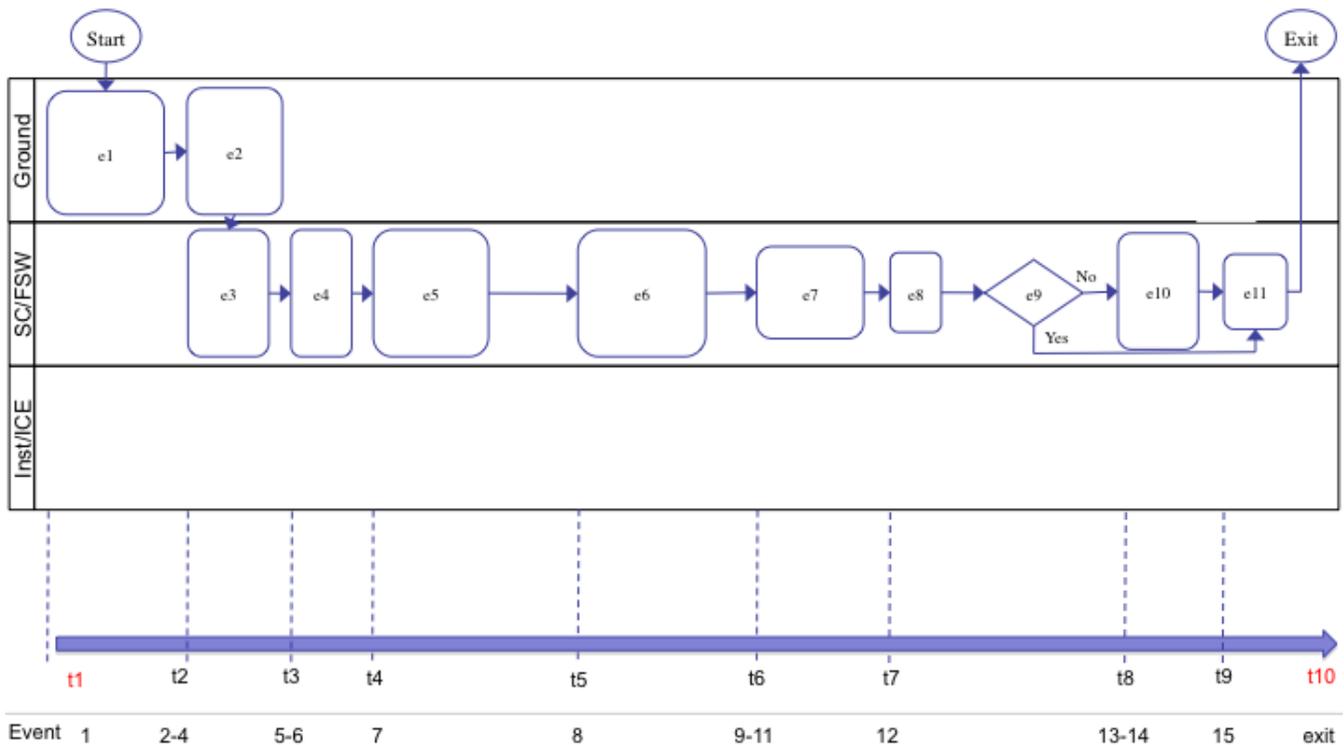


Figure 7. This is a sanitized version of SMAP's Activity diagram specifying the sequence of events during Launch.

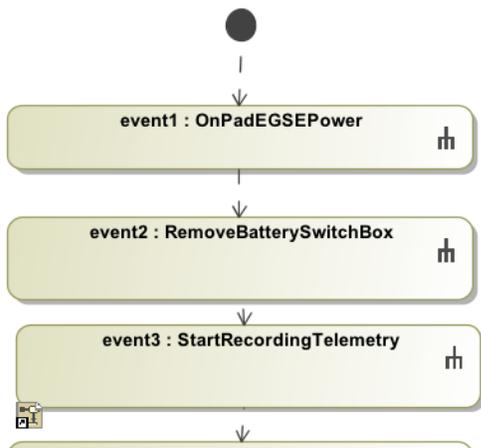


Figure 8. This is a snippet from our Scenario which amalgamates the information provided to us by SMAP into a single model. This snippet more closely resembles the Sequence spreadsheet, but each of these actions decomposes into an activity diagram that looks similar to SMAPs original activity diagram.

Scenario to Timeline is that the recursive transformation from Action to DurativeEvent can be stopped when either an Action has no associated Behavior or an Action has a Behavior that specializes the Template Behavior (specified at the Scenario Root). By that, we mean that the Scenario (in this case Launch Scenario) specifies that it is tracking a Template Behavior (in this case PowerTemplateActivity), which defines several DependentVariables that we may want to track over time (in this case *powerCBE*, *powerMargin*, *powerContingency*, and *powerAllocation*), and that all the

Behaviors in the Behavior Specification for each leaf component specialize that Template Behavior (in this case giving each of them the desired DependentVariable *powerCBE*). The rationale for the assumption that we can prune the transformation at Actions with Behaviors that specialize the Template Behavior is that a Behavior specializing the Template Behavior constrains the DependentVariable for a specified interval of time. As such, there is no useful additional information that can be gained by digging deeper into the Scenario's Activity tree. Elaboration of a Scenario as the project evolves can be done via a two step process: 1) remove the specialization of the Behavior to the Template Behavior and 2) add child Behaviors that each specialize the Template Behavior. An example Template Behavior is shown in Figure 9.

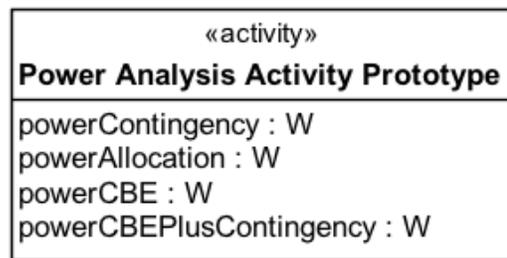


Figure 9. This is an example of a Template Behavior. It has several properties which can be tracked over time (DependentVariables). In this case, we tracked *powerCBE*.

After specifying the Scenario, we were able to use the Activity-To-Timeline-Instance transformation (discussed in Section 5) to export the scenario specification to a Timeline instance, allowing us to use our Timeline Solver (discussed

in Section 5) and Timeline Viewer (discussed in Section 5) to produce a full power analysis for SMAP for the specified Scenario. An example plot for a subsystem is shown in Figure 10. Note that the Scenario starts before $t = 0$ as $t = 0$ corresponds with the actual launch of the FlightSystem. The Scenario includes pre-launch activities.

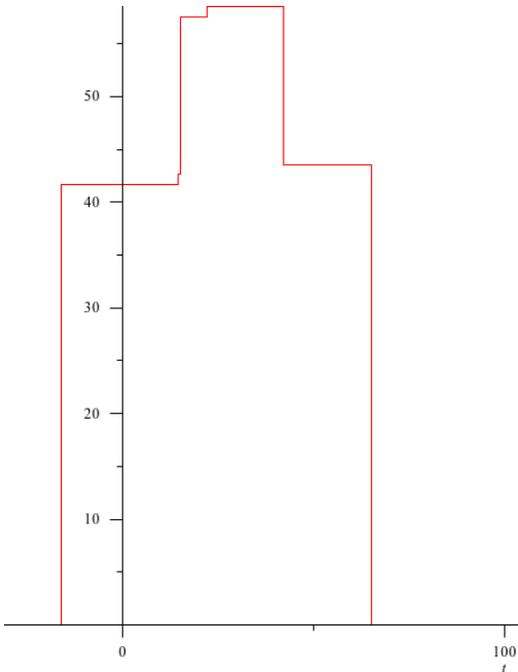


Figure 10. This plot shows a notional power profile for one of the SMAP’s subsystems during the Launch Scenario (power in watts over time in minutes).

Application for MOS/GDS Engineering (EPOXI)

The second part of this project has looked at the current structure for MOS/GDS tools to see how Timeline could be used to make our system better. We focused on the Extrasolar Planet Observation and Characterization (EPOXI) mission and how it used the planning tool Activity Plan Generator (APGEN) to orchestrate various power, telecommunications, thermal, trajectory, and attitude simulation tools to develop sequences.

APGEN, as the orchestrator, needs to be aware of the behavior of all of these tools. Each of the tools orchestrated by APGEN has its own unique behavior. Many of these tools has its own definition of time and how time steps are realized. For example, the Multi-Mission Power Analysis Tool (MMPAT) maintains power-related state information for each component in a system. When asked to step time and update the power-state information, MMPAT occasionally decides that it needs to break that time-step into multiple steps because the power information is changing too rapidly. In order to account for this, APGEN must loop on its call to MMPAT until MMPAT has stepped to the desired time.

Behavior Modeling for APGEN—Instead of modeling a single execution trace (as we did with SMAP), we wanted to use Timeline to model the entire behavior space of a tool. This involved taking C++ code and domain-specific language files representing APGEN’s orchestration of the different tools and transforming those into our Timeline constructs. Further discussion on this modeling effort will be deferred to a future

publication.

7. NEXT STEPS

Timeline Ontology Work

Timeline Specialization—The current Timeline Ontology is not explicit about what type of information is expected in the Timeline. Timelines currently constrain a `DependentVariable` over time. This constraint can be to a mathematical function, but it could instead be to a state string. Additionally, although it is possible to define a discrete Timeline of instant events representing some sampled signal, there is nothing formal to restrict a Timeline to contain only discrete sampled information. To remedy this, we propose adding at least the following types of Timelines to the ontology:

- **Discrete Event Timeline:** a Timeline in which all leaf-level events are `InstantEvents`.
- **Discrete State Timeline:** a Timeline in which all constraints constrain `DependentVariables` to a member of a set of strings defined in an enumeration for each Component.

Necessary Timeline augmentations—While modeling EPOXI’s use of APGEN, we found that in order to properly model behavior, we need to add additional constructs to the Timeline Ontology. Specifically, we need specialized Events for handling conditional branching (e.g. `If-Then-Else`) and a representation for loops (e.g. `For`, `While`, and `Do-While`). While these events are not necessary for describing traces of as-executed behavior, they are very necessary for describing the entire space of possible behaviors.

Timeline eXchange Infrastructure Work

Currently, the Timeline infrastructure we have created assumes a single `DependentVariable` type. Using this infrastructure, we can make a Timeline instance for `powerCBE` for each component in the system, but if we wanted to create a Timeline instance for `dataRateCBE`, we would have to recreate the same structure described in the Timeline instance. It would be better to allow for multiple `DependentVariable` types within the same Timeline instance so we would not have to separately track consistency between Timeline instance structures for the same “Scenario”. That is to say that instead of having a `LaunchPower` Timeline instance and a separate `LaunchDataRate` Timeline instance, we would prefer to have a single `Launch` Timeline instance that tracks both `powerCBE` and `dataRateCBE`.

TXI Tool Work

One assumption we made while developing the TXI tools was that all Constraints in a Timeline instance are purely mathematical. Our tools have no understanding of discrete “state” strings (e.g. “On”, “Off”, “Warmup”), although they are not precluded by the Timeline Ontology. The Timeline Solver would not be able to solve for the dependent variable over time (in this case `powerCBE`) for a component specified in this way, and as all components are related via the implicit physical decomposition constraints, we would not be able to solve for any parent component in that tree, all the way up to the `FlightSystem`. As the Timeline Solver would not be able to solve the timeline instance, the Timeline Viewer would not be able to plot the results.

This issue is currently not a big problem; however, as we plan to add explicit discrete state Timelines to the Timeline Ontology, we will need to update our tools accordingly.

Planned Tool Extensions

Currently, the only way to edit an existing Timeline instance is to manipulate the Timeline instance in Java code. We would like to instead create a Timeline Editor tool, a front-end to our Timeline instance (NamedIndividual) database that allows us to:

- view a Timeline instance's structure, relationships, and values; and
- manipulate a Timeline instance's structure, relationships, and values via insertion, deletion, and mutation of Timeline elements.

OWL Reasoner Integration

Currently, all our Timeline instances are assumed to be valid, but there is no rigorous check to ensure Timeline instance conformance with the Timeline Ontology. This check can be performed by using an OWL reasoner to validate the instances. We plan to integrate Pellet [9] into our system to perform these checks.

Future Direction

With the TXI currently storing information about power for systems, we would like to start adding in other state variables of interest, like DataRate, Mass, Attitude, Trajectory, etc. and linking those Timeline instances together. This has the potential to be a very powerful infrastructure for capturing time-varying information in a fully integrated way.

ACKNOWLEDGMENTS

This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

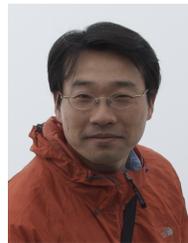
REFERENCES

- [1] A DAML Ontology of Time. [Online]. Available: <https://dspace.ist.utl.pt/bitstream/2295/40765/1/ADAMLOntologyOfTime.pdf>
- [2] Time Ontology in OWL. [Online]. Available: <http://www.w3.org/TR/owl-time/>
- [3] S. H. Chung and D. L. Bindschadler, "Timeline-based mission operations architecture: An overview," in *Proceedings of the 12th International Conference on Space Operations (SpaceOps 2012)*, Stockholm, Sweden, June 11–15 2012.
- [4] The Protégé Ontology Editor and Knowledge Acquisition System. [Online]. Available: <http://protege.stanford.edu/>
- [5] OWL - Semantic Web Standards. [Online]. Available: <http://www.w3.org/OWL/>
- [6] RDF - Semantic Web Standards. [Online]. Available: <http://www.w3.org/RDF/>
- [7] openRDF.org: Home. [Online]. Available: <http://www.openrdf.org/>
- [8] Maple 16 - Technical Computing Software for Engineers, Mathematicians, Scientists, Instructors and Students - Maplesoft. [Online]. Available: <http://www.maplesoft.com/products/maple/>
- [9] Pellet: OWL 2 Reasoner for Java. [Online]. Available: <http://clarkparsia.com/pellet/>

BIOGRAPHY



Kenneth Donahue is a member of the System Architectures and Behaviors Group at the Jet Propulsion Laboratory. He holds key roles on several projects spearheading Model-Based Systems Engineering at JPL. He is also the Radiometer Flight Software Cognizant Engineer on CHARM, a small flight project collaboration between JPL and Ames Research Center. He received his Bachelor of Science and Masters of Engineering from the Massachusetts Institute of Technology.



Seung Chung is the Technical Group Supervisor of the Modeling and Verification Group at the Jet Propulsion Laboratory. His group is responsible for developing and maintaining the tools used to verify and validate spacecraft command sequences and for developing the spacecraft models and the operational rules needed to verify and validate these sequences. He also holds leadership roles in architecting model-based approaches to systems engineering for both ground and flight systems, including the Europa mission concept studies. He received his Ph.D. in Autonomy from the Massachusetts Institute of Technology.