TEST-Documentation

Table of contents

Mission Modeler Guide

Introduction

This document is a guide to how to make use of current Aerie capabilities as of A29.0 delivery. AERIE is a new software system being developed by the MPSA element of Multi -mission Ground System and Services (MGSS), a subsystem of AMMOS (Advanced Multi -mission Operations System). AERIE will support activity planning, sequencing, and spacecraft analysis of mission operations. This guide will be updated as new features are added.

Aerie Overview

Aerie is a collection of loosely coupled services that support activity planning and sequencing needs of missions with modelling, simulation, scheduling and validation capabilities. Aerie will replace legacy MGSS tools including but not limited to APGEN, SEQGEN, MPS Editor, MPS Server, Slinc II / CTS and ULSGEN. A29.0 version of Aerie provides mainly the following:

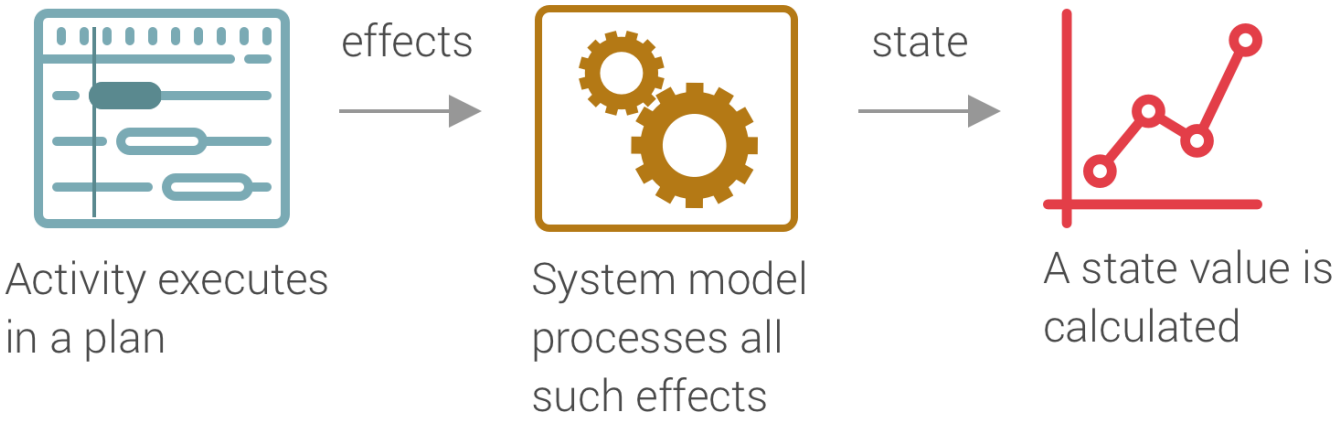
- Merlin adaptation framework offering a subset of APGEN capabilities
- Merlin web GUI for activity planning
- Merlin command line interface for activity planning
- Falcon smart command editor GUI

References

Aerie Software Requirements Document (Aerie SRD) - DOC-002388 Aerie Product Guide - DOC-002537 MGSS Implementation and Task Requirements - DOC -001455 JPL Software Development Rules 57653

Creating a Mission Model with Merlin Framework

A Merlin mission model can simulate a variety of states, governed by system models, and perturbed by executed activities. The below diagram summarizes the relationships between activities, system models, and states that are simulated over the course of a mission plan.



In order to complete this plan simulation, adaptation has to have activities with effect models, and system models that describe the response to these effects. A29.0 Merlin adaptation framework provides constructs to describe activities, system models and states. Scheduling, scheduling rules and constraints are not yet supported by the Merlin adaptation framework.

Creating A Mission Simulation Overview

Running an Aerie simulation requires an adaptation that describes effects of activities over modeled states, and a plan file that declares a schedule of activity instances with specified parameters. A plan file must be created with respect to an adaptation file, since activities in a plan and their parameters are validated against activity type definitions in the adaptation.

Aerie takes the adaptation and the plan file as inputs and executes a simulation. Simulation currently returns all states declared in the adaptation at a parameterized sampling rate. The simulation results format will be improved in upcoming releases. In later releases of Aerie will also return constraint violation results.

Here's a summary workflow to getting a simulation result from Aerie.

1. Install Aerie services following instructions on Product Guide
2. Create an Aerie adaptation following the instructions on Developing an Adaptation page.
3. Upload the adaptation to the adaptation service through Planning Web GUI or CLI-Command Line Interface.
4. Create a plan with the adaptation using again the Planning Web GUI or CLI
5. Trigger simulation via either interfaces listed above.

Details of each step are described in their respective pages.

Developing An Adaptation

An adaptation defines a set of activity types and states that make up a system model to be simulated by Merlin. To create an adaptation, the merlin-sdk must be included in your Java project. The merlin-sdk JAR file is available in Artifactory at this location.

Note that to access the download link for the merlin-sdk, you will need to log in to Artifactory. Once downloaded, decompress the file to your desired location. The merlin-sdk.jar file is available under the merlin-sdk folder. Include this JAR as a library in your Java project. Instructions how to include a library in a project for your Java IDE should be easy to find online.

Adaptation Class

Every adaptation must provide an adaptation class which satisfies the following:

1. Use `@Adaptation` annotation on the class, declaring adaptation name and version
2. Implement the `MerlinAdaptation` interface, supplying an Event type
3. Provide activity mappers by implementing the `getActivityMapper()` method
4. Provide a Querier by implementing the `makeQuerier()` method

Below is an example of an Adaptation class which uses custom ExampleEvent and ExampleQuerier types, which will be discussed in subsequent sections:

ExampleAdaptation Class

```
package gov.nasa.jpl.example;

@Adaptation(name="example", version="0.1")
public class ExampleAdaptation implements MerlinAdaptation<ExampleEvent> {

   @Override
   public ActivityMapper getActivityMapper() {
      try {
         return ActivityMapperLoader.loadActivityMapper(ExampleAdaptation.class);
      } catch (ActivityMapperLoader.ActivityMapperLoadException e) {
         e.printStackTrace();
         return null;
      }
   }

   @Override
   public <T> Querier<T, ExampleEvent> makeQuerier(final SimulationTimeline<T, BananaEvent> database) {
      return new ExampleQuerier<>(database);
   }
}
```

Events

As previously mentioned, Merlin adaptations define activity types which influence states during a simulation. Events are used to represent these influences. As there are different types of possible Events that can occur, we use the visitor pattern to define a uniform Event type that can be used in an adaptation. We begin by defining an Event class, such as the one shown below:

```
public abstract class ExampleEvent {
   private ExampleEvent() {}

   public abstract <Result> Result visit(ExampleEventHandler<Result> visitor);

   public static ExampleEvent independent(final IndependentStateEvent event) {
      Objects.requireNonNull(event);
      return new ExampleEvent() {
         @Override
         public <Result> Result visit(final ExampleEventHandler<Result> visitor) {
            return visitor.independent(event);
         }
      };
   }

   public static ExampleEvent activity(final ActivityEvent event) {
      Objects.requireNonNull(event);
      return new ExampleEvent() {
         @Override
         public <Result> Result visit(final ExampleEventHandler<Result> visitor) {
            return visitor.activity(event);
         }
      };
   }

   public final Optional<IndependentStateEvent> asIndependent() {
      return this.visit(new DefaultExampleEventHandler<>() {
         @Override
         public Optional<IndependentStateEvent> unhandled() {
            return Optional.empty();
         }

         @Override
         public Optional<IndependentStateEvent> independent(IndependentStateEvent event) {
            return Optional.of(event);
         }
      });
   }

   public final Optional<ActivityEvent> asActivity() {
      return this.visit(new DefaultExampleEventHandler<>() {
         @Override
         public Optional<ActivityEvent> unhandled() {
            return Optional.empty();
         }

         @Override
         public Optional<ActivityEvent> activity(final ActivityEvent event) {
            return Optional.of(event);
         }
      });
   }
}
```

In short, the use of the visitor pattern here allows any event to be easily interpreted as the appropriate type. In this example, there are two types of events, independent state and activity events.

The careful observer may have noticed that there are two types referenced in the above example which have not yet been defined: `ExampleEventHandler` and `DefaultExampleEventHandler`. These types provide the second half of the visitor pattern, the actual visitor that interprets events. Both are shown below:

```
public interface ExampleEventHandler<Result> {
   Result independent(IndependentStateEvent event);
   Result activity(ActivityEvent event);
}
```

```
public interface DefaultExampleEventHandler<Result> extends ExampleEventHandler<Result> {
   Result unhandled();

   @Override
   default Result independent(IndependentStateEvent event) {
      return unhandled();
   }

   @Override
   default Result activity(ActivityEvent event) {
      return unhandled();
   }
}
```

The `ExampleEventHandler` interface is a barebones visitor definition, declaring a method for each type of Event that may be visited. `DefaultExampleEventHandler` extends `ExampleEventHandler` by adding an `unhandled()` method, and defining a default implementation of the `independent()` method that calls it. This allows instances of `ExampleEventHandler` to be created on the fly without having to define every method. This is especially useful when creating an event handler for a known subset of event types.

Together, the above class and interfaces define an adaptation's `Event`. In Aerie 0.4, these can be copied with little modification for a custom adaptation.

Querier

With your adaptation's event types defined, you can now define a Querier. The purpose of the Querier is to provide a connection between Merlin's simulation engine and your adaptation logic. The Querier has four basic responsibilities:

1. Run activities in a provided context
2. Provide state names
3. Determine state values from an event history
4. Determine constraint violations from an event history

In Aerie 0.4, this is a bit complex, but we provide a template to make implementing your Querier as simple as possible. Below is an example:

ExampleQuerier

```
public class ExampleQuerier<T> implements MerlinAdaptation.Querier<T, ExampleEvent> {
  private final static DynamicCell<ReactionContext<?, Activity, ExampleEvent>> reactionContext = DynamicCell.create();
  private final static DynamicCell<ExampleQuerier<?>.StateQuerier> stateContext = DynamicCell.create();

  public static final Function<String, StateQuery<SerializedParameter>> query = (name) ->
      new DynamicStateQuery<>(() -> stateContext.get().getRegisterQuery(name));
  public static final ActivityModelQuerier activityQuerier =
      new DynamicActivityModelQuerier(() -> stateContext.get().getActivityQuery());

  public static final ReactionContext<?, Activity, ExampleEvent> ctx = new DynamicReactionContext<>(() -> reactionContext.get());

  private final ActivityMapper activityMapper;
  private final Query<T, ExampleEvent, ActivityModel> activityModel;
  private final Set<String> stateNames = new HashSet<>();
  private final Map<String, Query<T, ExampleEvent, RegisterState<SerializedParameter>>> settables = new HashMap<>();
  private final Map<String, Query<T, ExampleEvent, RegisterState<Double>>> cumulables = new HashMap<>();

  public ExampleQuerier(final ActivityMapper activityMapper, final SimulationTimeline<T, ExampleEvent> timeline) {
    this.activityMapper = activityMapper;

    this.activityModel = timeline.register(
        new ActivityEffectEvaluator().filterContramap(ExampleEvent::asActivity),
        new ActivityModelApplicator());

    // Register a Query object for each settable state
    for (final var entry : ExampleStates.factory.getSettableStates().entrySet()) {
      final var name = entry.getKey();
      final var initialValue = entry.getValue();

      if (this.stateNames.contains(name)) throw new RuntimeException("State \"" + name + "\" already defined");
      this.stateNames.add(name);

      final var query = timeline.register(
          new SettableEffectEvaluator(name).filterContramap(ExampleEvent::asIndependent),
          new SettableStateApplicator(initialValue));

      this.settables.put(name, query);
    }

    // Register a Query object for each cumulable state
    for (final var entry : ExampleStates.factory.getCumulableStates().entrySet()) {
      final var name = entry.getKey();
      final var initialValue = entry.getValue();

      if (this.stateNames.contains(name)) throw new RuntimeException("State \"" + name + "\" already defined");
      this.stateNames.add(name);

      final var query = timeline.register(
          new CumulableEffectEvaluator(name).filterContramap(ExampleEvent::asIndependent),
          new CumulableStateApplicator(initialValue));
```

```
            this.cumulables.put(name, query);
        }
    }

    @Override
    public void runActivity(ReactionContext<T, Activity, ExampleEvent> ctx, String activityId, Activity activity) {
        reactionContext.setWithin(ctx, () ->
                stateContext.setWithin(new StateQuerier(ctx::now), () ->
                        activity.modelEffects()));
    }

    @Override
    public Set<String> states() {
        return Collections.unmodifiableSet(this.stateNames);
    }

    @Override
    public SerializedParameter getSerializedStateAt(String name, History<T, ExampleEvent> history) {
        if (this.settables.containsKey(name)) return this.settables.get(name).getAt(history).get();
        else if (this.cumulables.containsKey(name)) return SerializedParameter.of(this.cumulables.get(name).getAt(history).get());
        else throw new RuntimeException("State \"" + name + "\" is not defined");
    }

    @Override
    public List<ConstraintViolation> getConstraintViolationsAt(History<T, ExampleEvent> history) {
        final List<ConstraintViolation> violations = new ArrayList<>();

        final var stateQuerier = new StateQuerier(() -> history);
        for (final var violableConstraint : ExampleStates.violableConstraints) {
            // Set the constraint's getWindows method within the context of the history and evaluate it
            final var violationWindows = stateContext.setWithin(stateQuerier, violableConstraint::getWindows);
            if (!violationWindows.isEmpty()) {
                violations.add(new ConstraintViolation(violationWindows, violableConstraint));
            }
        }

        return violations;
    }

    public StateQuery<SerializedParameter> getRegisterQueryAt(final String name, final History<T, ExampleEvent> history) {
        if (this.settables.containsKey(name)) return this.settables.get(name).getAt(history);
        else if (this.cumulables.containsKey(name)) return StateQuery.from(this.cumulables.get(name).getAt(history), SerializedParameter::of);
        else throw new RuntimeException("State \"" + name + "\" is not defined");
    }

    public final class StateQuerier {
        private final Supplier<History<T, ExampleEvent>> historySupplier;

        public StateQuerier(final Supplier<History<T, ExampleEvent>> historySupplier) {
            this.historySupplier = historySupplier;
        }

        public StateQuery<SerializedParameter> getRegisterQuery(final String name) {
            return ExampleQuerier.this.getRegisterQueryAt(name, this.historySupplier.get());
        }

        public ActivityModelQuerier getActivityQuery() {
            return ExampleQuerier.this.activityModel.getAt(this.historySupplier.get());
        }
    }
}
```

It is not necessary to understand the details of this file, since our template should get you set up and you'll need not worry about it again. Thus, if you are uninterested you may skip to the next section.

If you are still reading, then let's take a look at our `ExampleQuerier`. Starting at the top, we have two `DynamicCell` context variables. In short, a `DynamicCell` is a container for something, in this case context, that may change. This allows us to pass context to adaptation logic without having to pass it through the callstack by inserting it into the cell prior to calling the adaptation. Throughout the Querier you can see `setWithin()` called on the `DynamicCells`. This is what sets the value in the `DynamicCell` before running the adaptation logic that needs the context. In the case of activity models, the `ctx` variable declared farther down provides an interface to reach the current context.

The `query` field provides an interface into the `StateQuerier` inner class defined at the bottom, which provides access to the `stateContext` to activities and states. Given an event history, the `StateQuerier` provides the ability to get a state value, as well as the ability to determine windows when a condition is met by some state. After the previously mentioned `ctx` variable are activity and state information variables. These are instantiated by the constructor, and should hold the activity mapper and model tools, as well as all the settable and cumulable states. The constructor itself just populates these variables using the states class that should be defined (see here).

The `runActivity()` method takes a `ReactionContext` as well as an activity instance to run. The method body sets the `reactionContext` and provides a lambda expression. The lambda then sets the `stateContext` and runs the activity model. With both `DynamicCell`s populated, the activity can access the `reactionContext` through the public `ctx` variable, or the `stateContext` through the public `query` variable.

The `states()` method returns a set of the available state names. Following that is `getSerializedStateAt()`, which provides the value of a named state given a history. The `getConstraintViolationsAt()` method processes an event history to determine any constraint violations. It is worth noting that to determine constraint violations, only the `stateContext` is needed, so only the `stateContext` is given a value. Finally, `getRegisterQueryAt()` provides a `StateQuery` over a given parameter. This is used by the `StateQuerier` inner class for the `getRegisterQuery()` method.

Building a Mission Model

With your basic adaptation setup, you will be ready to begin defining your mission model. A mission model consists of three main pieces: activities, states and constraints. In a typical adaptation, states are used to track available resources, and other system state. Activities define actions that the system can perform, which usually have effects on system state. Constraints serve a variety of purposes, including flagging undesirable (or unacceptable) conditions when they occur, and determining appropriate windows of opportunity for scheduling activities.

Uploading an Adaptation

When your adaptation is ready to be uploaded, the first step is to package it into a JAR file. This can be done manually, or using your IDE's built-in tools, though it is necessary to include all dependencies in the JAR. Be sure to include the contents of the resources folder as well, as the META-INF directory should contain the services folder that tells Merlin about your adaptation class.

Once a JAR is acquired, it must be uploaded the Aerie through one of our interfaces (merlin-cli or the GUI). When uploading an adaptation, in addition to the JAR file you must supply a name, version, mission and owner. Currently, it is vital that the name and version match those declared in the `@Adaptation` annotation on the adaptation class. If this is not the case, the adaptation will fail to load into Merlin properly, and the upload request will be rejected.

Activities

- To include: workarounds for activity and parameter mappers

The mission system's behavior is modeled as a series of activities that emit discrete events. These events are manifested by the real mission system as the schedule of tasks of ground based assets and a spacecraft's onboard sequences, commands, or flight software. Activities in Merlin are entities whose role is to emit stimuli (events) to which the mission model will react. Activities can therefore describe the relation: "when this activity occurs, this kind of thing should happen".

Activity Type are prototypes for activity instances that will be executed in a plan. Activity Type are java classes that implement the Activity interface provided by the Merlin framework, include a default constructor, and optional overloaded constructors. When compiled, each Activity Type will have its own .java file. Activity types can be organized into hierarchical packages, for example as `gov.nasa.jpl.europa.clipper.gnc.TCMActivity`

Activity Type consist of:

- metadata
- parameters that describe the range in execution and effects of the activity
- effect model that describes how the system will be perturbed when the activity is executed.

Activity Annotation

An activity type definition should be annotated with the `@ActivityType` tag. An activity type is declared with its name using the following annotation:

```
@ActivityType(name="TurnInstrumentOff", generateMapper=True)
```

By doing so, the Merlin annotation processor can discover all activity types declared in the mission model, validate that activity type names are unique. The `generateMapper` flag tells Merlin to generate an activity mapper for the activity. If this is left out, or set to false, you will need to provide your own custom activity mapper. See here for more information on activity mappers.

Activity Metadata

Metadata of activities are structured such that the Merlin annotation processor can extract this metadata given particular keywords. Currently, the Merlin annotation processor recognizes the following tags: `contact, subsystem, brief_description,` and `verbose_description.`

These metadata tags are placed in a JavaDocs style comment block above the Activity Type to which they refer. For example:

```
/**
 * @subsystem Data
 * @contact mkumar
 * @brief_description A data management activity that deletes old files
 */
```

These tags are processed, at compile time, by the annotation processor to create documentation for the Activity types that are described in the mission model.

Activity Parameters

Parameters for activity types define how much an activity execution can vary. These parameters are used to determine the effects of the activity, as well as it’s duration, decomposition and expansion into commands.

```
/**
 * The bus power consumed by the instrument while it is turned on measured in Watts
 */
@Parameter
public double instrumentPower_W = 100.0;
```

The annotation processor is similarly used to extract and generate serialization code for parameters of activity types. The annotation processor also allows authors of a mission model to create mission specific parameter types, ensuring that they will be recognized by the Merlin framework.

Parameters of an activity can be validated and restricted by providing a `validateParameters()` method:

```
@Override
public List<String>  validateParameters() {
  final List<String> failures = new ArrayList<>();
  if (instrumentPower <=  0.0 || instrumentPower >=  1000.0) {
    failures.add("data rate must be positive and greater than 0");
  }
  return failures;
}
```

Activity Effects

Effects of activity types should be defined in the `modelEffects()` method of the activity. The `modelEffects()` method is called by the simulation engine when the activity is executed.

The Merlin simulation engine is decoupled from the other areas of Merlin, such as states and activities. The simulation context is an implicit (behind the scenes) provider of semantic simulation control and provides a number of functions which allow for specific manipulation of activities within the simulation:

- `delay(duration)`: Delay the currently-running activity for the given duration. On resumption, it will observe effects caused by other activities over the intervening timespan.
- `waitForActivity(activityId)`: Delay the currently-running activity until the activity with specified ID has completed. On resumption, it will observe effects caused by other activities over the intervening timespan.
- `waitForChildren()`: Delay the currently-running activity until all child activities have completed. On resumption, it will observe effects caused by other activities over the intervening timespan.
- `spawn(activity)`: Spawn a new activity as a child of the currently-running activity at the current point in time. The child will initially see any effects caused by its parent up to this point. The parent will continue execution uninterrupted, and will not intially see any effects caused by its child.
- `spawnAfter(duration, activity)`: Asynchronously spawn a new activity as a child of the currently-running activity at the specired duration after the current point in time. The child will initially see any effects caused by its parent up to that point. The parent will continue execution from the current time point uninterrupted, and will not initially see any effects caused by its child.
- `call(activity)`: Spawn a new activity as a child of the currently-running activity at the current point in time. The child will initially see any effects caused by its parent up to this point. The parent will halt execution until the child activity has completed. This is equivalent to calling `waitForActivity(spawn(activity))`.

This context is provided from the Merlin simulation engine to an adaptation's Querier (seehere) such that it can be imported directly into your activity models from your Querier. Below is an example of an activity which imports the `ctx` context variable from the Querier shown here.

```
import static gov.nasa.jpl.example.states.ExampleQuerier.ctx;
import static gov.nasa.jpl.example.states.ExampleStates.batteryCapacity;

@ActivityType(name="RunHeater", generateMapper=true)
public class RunHeater implements Activity {

   private static final int energyConsumptionRate = 1000;

   @Parameter
   public long durationInSeconds;

   @Override
   public void modelEffects() {
      final double totalEnergyUsed = this.durationInSeconds*energyConsumptionRate;

      ctx.spawn(new PowerOnHeater());
      batteryCapacity.add(-totalEnergyUsed);
      ctx.spawnAfter(Duration.of(this.durationInSeconds, TimeUnit.SECONDS), new PowerOffHeater());
      ctx.waitForChildren();
   }
}
```

Since the `ctx` context and `batteryCapacity` state variables are statically imported, they can be referred to directly by name in the `modelEffects()` method. This activity first places a `PowerOnHeater` activity at the start of the activity. Next the total energy used by

running a heater for a parameterized duration is subtracted from the batteryCapacity state. Next the `PowerOffHeater` activity is queued up to occur after the run duration. Finally, the activity waits for its children, ensuring that the duration of this activity covers it's children.

A Note about Decomposition

In Merlin mission models, decomposition of an activity is not an independent method, rather it is defined within the `modelEffects()` method by means of invoking child activities. These activities can be invoked using the `call()` method, where the rest of the modelEffects code waits for the child activity to complete; or using the `spawn()` and `spawnAfter()` methods, where the modelEffects continues to execute without waiting for the child activity to complete. These two methods allow any arbitrary serial and parallel arrangement of child activities. This approach replaces duration estimate based wait calls with event based waits. Hence, this allows for not keeping track of estimated durations of activities, while also improving the readability of the activity procedure as a linear sequence of events. Merlin supports a maximum duration parameter which can be used to detect modeling / simulation errors, as well as for display purposes. After a simulation run, simulated durations can be used for display purposes.

If desirable, users can choose to follow APGEN / Blackbird paradigm by simply spawning all child activities at the beginning of the modelEffects() method, and then posting effects interleaved with `waitFor(duration)` method, where the duration is the estimate for child activity durations.

Activity Mappers

- To include: Documentation on `SerializedValue` for assistance with creating custom Activity Mappers. We should expand on `ValueSchema` as well, though we will likely want to change the way we approach the topic once we add support for generating Activity Mappers that use custom Value Mappers.

What is an Activity Mapper

An Activity Mapper is a Java class that both implements the `ActivityMapper` interface, and contains the `ActivitiesMapped` annotation. It is required that each Activity Type in an adaptation have an associated Activity Mapper, to provide provide three capabilities:

- Describe the structure of an activity type
- Serialize instances of associated Activity Types
- Deserialize instances of associated Activity Types

`@ActivitiesMapped` Annotation

The `@ActivitiesMapped` annotation tells what Activity Types the associated Activity Mapper provides the aforementioned capabilities for. Below is an example of the annotation as it would appear in an adaptation for an Activity Mapper called `ExampleMapper` that maps a single Activity Type called `ExampleActivity` (It is important to note that the Activity Types listed in the annotation be denoted by their class name, which may sometimes differ from the actual Activity Type name).

```
@ActivitiesMapped({ExampleActivity.class})
public class ExampleMapper implements ActivityMapper { ... }
```

Activity Schema

Each Activity Mapper must describe the structure of its associated Activity Types by implementing the `getActivitySchemas()` method whose signature is shown below:

```
Map<String, Map<String, ValueSchema>> getActivitySchemas();
```

The `Map` returned by this method should contain a schema for each associated Activity Type, indexed by Activity Type name. Each activity schema is represented by another `Map`, supplying parameter names as keys, and the corresponding `ValueSchema`s as values.

Serialization

Activity Mappers must supply a `serialize()` method as specified by the following signature:

```
Optional<SerializedActivity> serializeActivity(Activity activity);
```

The `Optional` returned by this method should be empty if the provided `Activity` is not an instance of an Activity Type mapped by this Activity Mapper. However, if the provided `Activity` is indeed an instance of one of the Activity Types the Activity Mapper handles, then a `SerializedActivity` representing the instance should be returned.

Deserialization

The `deserialize()` method must also be provided by Activity Mappers, and is described by the following method signature:

```
Optional<Activity> deserializeActivity(SerializedActivity serializedActivity);
```

The `Optional` returned by this method should be empty if no instance for any of the associated Activity Types can be constructed from the provided `SerializedActivity`. Otherwise, an instance of the associated Activity Type that fits the provided `SerializedActivity` should be provided.

It is important to note that the result of an Activity Mapper's `serialize()` method be accepted by the mapper's `deserialize()` method, and vice-versa.

Creating Activity Mappers

Even if one understands what an Activity Mapper should contain, it is another challenge entirely to actually create one. In many cases, it may be enough to simply have Merlin generate one for Activity Types. If this is the case, `generateMapper` should be set to `true` in the `@ActivityType` annotation for the Activity Type (this is set to `false` by default).

Currently, Merlin can generate Activity Mappers for Activity Types which contain only supported parameter types. This means that for Activity Types containing custom parameter types, adaptation engineers will need to supply custom activity mappers (we are planning to add support for custom activity mappers in the future, though some work by the adaptation engineer will still be required).

Using Custom Parameter Types in Activity Types

If a custom parameter type is needed in an Activity Type, a custom mapper must be created. This is a two step process:

1. Custom activity mapper must be created for the activity in which you want to add the custom parameter.
2. Custom value mapper has to be created. To expedite the first mapper generation, adaptation engineers can rely on the automatic activity mapper generator by excluding the custom parameters in a first pass, and then manually edit the mapper to include the custom parameters. Entities with custom mappers should not include annotations to avoid conflict between automatic generation process.

To create a custom Activity Mapper, a Java class meeting the criteria for Activity Mappers should be created. A skeleton for all Activity Mappers is given below:

```
@ActivitiesMapped({ExampleActivity.class})
public class ExampleMapper implements ActivityMapper {

    @Override
    public Map<String, Map<String, ValueSchema>> getActivitySchemas() {
        ...
    }

    @Override
    public Optional<Activity> deserializeActivity(SerializedActivity serializedActivity) {
        ...
    }

    @Override
    public Optional<SerializedActivity> serializeActivity(Activity activity) {
        ...
    }
}
```

A custom Activity Mapper may be constructed from scratch, though one can get a head start by generating a partial mapper for an Activity Type. To do this, temporarily comment out any `@Parameter` annotations for custom parameters, and add `generateMapper=true` to the `@ActivityType` annotation, then run the Merlin annotation processor. This will generate a mapper for the Activity Type that handles all supported parameter types. The adaptation engineer can then copy this generated mapper to their source folder and add logic for custom parameter types, using the generated code as a guide. Note that when using this procedure, the `@generated` annotation should be removed from the Activity Mapper, and the temporary changes to the Activity Type should be removed.

Value Mappers

In creating an Activity Mapper, it may be useful to use Value Mappers. A `ValueMapper` is similar to an `ActivityMapper`, but can be used to focus on a single Activity Parameter. Properly used, Value Mappers can make Activity Mappers quite simple, handling all the dirty work at the parameter level. In fact, Merlin's generated Activity Mappers work exclusively using Value Mappers.

One of the most convenient things about using Value Mappers is the fact that Merlin comes with them already defined for all supported parameter types. Furthermore, Value Mappers for combinations of supported types can easily be created by passing one `ValueMapper` into another during instantiation.

Although we provide Value Mappers for supported parameter types, it is entirely acceptable to create custom Value Mappers for use in custom Activity Mappers. This can be done by writing a Java class which implements the `ValueMapper` interface. Below is a Value Mapper for an apache `Vector3D` type as an example:

```java
public class Vector3DValueMapper implements ValueMapper<Vector3D> {

 @Override
 public ValueSchema getValueSchema() {
  return ValueSchema.ofSequence(ValueSchema.REAL);
 }

 @Override
 public Result<Vector3D, String> deserializeValue(final SerializedValue serializedValue) {
  return serializedValue
     .asList()
     .map(Result::<List<SerializedValue>, String>success)
     .orElseGet(() -> Result.failure("Expected list, got " + serializedValue.toString()))
     .match(
        serializedElements -> {
          if (serializedElements.size() != 3) return Result.failure("Expected 3 components, got " + serializedElements.size());
          final var components = new double[3];
          final var mapper = new DoubleValueMapper();
          for (int i=0; i<3; i++) {
            final var result = mapper.deserializeValue(serializedElements.get(i));
            if (result.getKind() == Result.Kind.Failure) return result.mapSuccess(_left -> null);

            // SAFETY: `result` must be a Success variant.
            components[i] = result.getSuccessOrThrow();
          }
          return Result.success(new Vector3D(components));
        },
        Result::failure
     );
 }

 @Override
 public SerializedValue serializeValue(final Vector3D value) {
  return SerializedValue.of(
     List.of(
        SerializedValue.of(value.getX()),
        SerializedValue.of(value.getY()),
        SerializedValue.of(value.getZ())
     )
  );
 }
}
```

Example Activity Mapper

Below is an example of an Activity Type and its Activity mapper for reference:

Activity Type

```java
@ActivityType(name=RunHeaters)
public class RunHeatersActivity {

   @Parameter
   public double heatDuration;

   @Parameter
   public int heatIntensity;

   @Override
   public void modelEffects() { ... }
}
```

Activity Mapper

```java
@ActivitiesMapped({RunHeatersActivity.class})
public class RunHeatersMapper implements ActivityMapper {

   // Instantiate a ValueMapper for each parameter of the RunHeater activity
   // These allow us to delegate to ValueMappers for parameter-specific work
   private static final ValueMapper<Double> mapper_heatDuration = new NullableValueMapper<>(new DoubleValueMapper());
   private static final ValueMapper<Integer> mapper_heatIntensity = new NullableValueMapper<>(new IntegerValueMapper());

   @Override
   public Map<String, Map<String, ValueSchema>> getActivitySchemas() {

      // Instantiate the Activity Schemas map
      Map<String, Map<String, ValueSchema>> activitySchemas = new HashMap<>();

      // Build the Activity Schema for the RunHeaters activity
      Map<String, ValueSchema> runHeatersSchema = new HashMap<>();
      runHeatersSchema.put("heatDuration", ValueSchema.DOUBLE);
      runHeatersSchema.put("heatIntensity", ValueSchema.INTEGER);

      // Populate the Activity Schemas map
      // Note that the Activity Type name is used for the Activity Schema map
      // instead of the Activity Type class
      activitySchemas.put("RunHeaters", runHeatersSchema);

      return activitySchemas;
   }

   @Override
   public Optional<Activity> deserializeActivity(SerializedActivity serializedActivity) {
      // If activity is not the supported type, return empty optional
      if (!serializedActivity.getTypeName().equals("RunHeaters")) {
        return Optional.empty();
      }

      // Create the Activity Instance and set parameter values if present
      final var activity = new RunHeatersActivity();
      for (final var entry : serializedActivity.getParameters().entrySet()) {
         switch (entry.getKey()) {
            case "heatDuration":
               activity.heatDuration = this.mapper_heatDuration.deserializeValue(entry.getValue()).getSuccessOrThrow();
               break;
            case "heatIntensity":
               activity.heatIntensity = this.mapper_heatIntensity.deserializeValue(entry.getValue()).getSuccessOrThrow();
               break;
            default:
               // Unexpected parameter key present, return empty optional
               return Optional.empty();
         }
      }

      return Optional.of(activity);
   }

   @Override
   public Optional<SerializedActivity> serializeActivity(Activity activity) {

      // If activity is not suppoorted, return empty Optional
      if (!(activity instanceof RunHeatersActivity)) return Optional.empty();

      final RunHeatersActivity activity = (RunHeatersActivity)activity;

      final var parameters = new HashMap<String, SerializedValue>();
      parameters.put("heatDuration", this.mapper_heatDuration.serializeValue(activity.heatDuration));
      parameters.put("heatIntensity", this.mapper_heatIntensity.serializeValue(activity.heatIntensity));

      return Optional.of(new SerializedActivity(ACTIVITY_TYPE, parameters));
   }
}
```

States & Models

Merlin States

In Merlin, a state is any value that is tracked during the course of a simulation. Merlin states can be used for a myriad of purposes, including tracking of finite resources, various geometric attributes, ground and flight events, spacecraft states that are altered by activities, and more. In Aerie 0.4, we provide two types of states that can be used in adaptations: settable and cumulable.

Settable states are states whose value can be changed by setting it directly to a desired value. An very common example of a settable state would be a spacecraft or instrument mode. Cumulable states are states whose value can be changed by adding a value to the current value. Examples of cumulable states include remaining battery capacity and volume in a data channel.

Creating States

To create states in Merlin, the first thing you will need is an instance of ou`IndependentStateFactory`. This factory hides away the details of settable and cumulable states so adaptation engineers only need declare the states they want to include in their adaptation. Let's take a look at an example state declaration class:

ExampleStates

```
import static gov.nasa.jpl.example.states.ExampleQuerier.activityQuerier;
import static gov.nasa.jpl.example.states.ExampleQuerier.query;
import static gov.nasa.jpl.example.states.ExampleQuerier.ctx;

public class ExampleStates {
    private static final ActivityTypeStateFactory activities = new ActivityTypeStateFactory(activityQuerier);
    public static final IndependentStateFactory factory = new IndependentStateFactory(query, (ev) -> ctx.emit(ExampleEvent.independent(ev)));

    public static final SettableState<Boolean> heaterPoweredOn = factory.bool("heaterPoweredOn", false);
    public static final DoubleState batteryCapacity = factory.cumulative("batteryCapacity", 1000000.0);

    public static final List<ViolableConstraint> violableConstraints = List.of(
            //...
    );
}
```

Here the `activities` and `factory` variables are created using the `activityQuerier`, `query` and `ctx` variables from the Querier shown on the Developing an Adaptation page. The `activities` variable can be used to constructactivity constraints.With the factory instantiated, states can be created by simply calling the appropriate method for the desired state type. References to these states are kept `public static final` so that activity models may easily access states while keeping them safe from reassignment, which would surely cause issues. The `violableConstraints` list at the bottom should be used to declare`ViolableConstraints`, for which more information can be found on the Constraints page.

It may be desirable, especially for large adaptations, to organize state declarations into separate files. In this case, you should keep a "master" state file, which instantiates the `IndependentStateFactory`. Each state declaration file should then refer to the master factory when declaring states, thus all states will be registered with the factory, and be tracked during simulation runs.

With an instance of an`IndependentStateFactory` in hand, states can now be added to your adaptation. In this section we will go over the types of states available in Aerie 0.4:

Cumulable States

In Aerie 0.4, cumulable states must have`Double` values. These can be created by supplying a name and initial value to the factory's `cumulative()` method:

```
public static final DoubleState batteryCapacity = factory.cumulative("batteryCapacity", 1000000.0);
```

Settable States

Settable states can take on a variety of types. Each can be created by calling the appropriate factory method, and supplying a name and initial value. Examples showcasing the appropriate method for each type are provided below.

String value

```
public static final SettableState<String> spacecraftMode = factory.string("spacecraftMode", "idle");
```

Integer value

```
public static final SettableState<Integer> samplesAcquired = factory.integer("samplesAcquired", 0);
```

Double value

```
public static final SettableState<Double> solarArrayAngle = factory.real("solarArrayAngle", 0.0);
```

Boolean value

```
public static final SettableState<Boolean> heaterPoweredOn = factory.bool("heaterPoweredOn", false);
```

Enumerated value

```
public enum InstrumentMode { ON, OFF }
public static final SettableState<InstrumentMode> instrumentState = factory.enumerated("instrumentState", InstrumentMode.OFF);
```

Custom Settable States

In addition to the pre-defined settable state types, `IndependentStateFactory` provides a `settable()` method to define a custom settable state. This method works similarly to the above factory methods, but it takes an additional parameter, `ValueMapper<T> mapper`. This method allows adaptation engineers to provide a `ValueMapper` (documented here) to create a settable state of a custom type. The following example demonstrates how to create such a state, assuming the adaptation engineer has defined a `CustomType` class along with a `CustomTypeValueMapper`:

```
public static final SettableState<CustomType> customState = factory.settable("customState", new CustomType(...), new CustomTypeValueMapper());
```

Custom Settable State Example

One example of a custom settable state exists in the sample-adaptation: the `cameraPointing` state. The `cameraPointing` state is a settable `Vector3D` as defined by `org.apache.commons.math3.geometry.euclidean.threed.Vector3D`. Using a custom value mapper, `Vector3DValueMapper`, the `cameraPointing` state is defined according to the specification given above, as shown below:

```
public static final SettableState<Vector3D> cameraPointing = factory.settable("cameraPointing", new Vector3D(1, 0, 0), new Vector3DValueMapper());
```

In this case, the `Vector3DValueMapper` exists in the merlin contrib module (where independent states are defined as well), so it can be imported from there. This may serve as an example, should a new custom `ValueMapper` be required.

Creating Plans

Plans can be created via Planning Web Gui or by uploading a JSON plan file through CLI. Instructions on these interfaces can be found in their respective pages.

A sample JSON Plan file format can be found in theUser Guide Appendix

Constraints

Constraints

Introduction

In general, constraints are used to describe behavior, typically undesired, and determine when in a schedule that behavior is occurring. This behavior can be simple, such as the following:
`The battery state of charge should never be below 30% of its full capacity at all times.`

They can also be used to express more complicated behavior, such as the following:
`When the S/C is less than 0.5AU from the Sun, the maximum rate in the x direction should not exceed 2mrad/sec, the camera instrument should not be on, and a downlink activity should not occur.`

This section describes how to represent these types of behavior and check when in a schedule this behavior is occurring with using the tools in the `Merlin-SDK`.

Atomic Constraints

`Atomic Constraints` are constraints that describe *a* condition for *one* State or *one* Activity. These constraints can be used as building blocks to create `Compound Constraints`. Alternatively, Atomic Constraints can be thought of as constraints that cannot be "broken down" any further.

For example, consider an adaptation that includes the following states and activities:

- `batterySOC`: a `State` that represents battery state of charge as a percentage
- `distanceFromSun`: a State that represents the S/C distance from the sun in AU
- `cameraStatus`: a State that represents the camera status as a String with the following expected values: `on, off, standby`
- `downlinkData`: an `Activity` that represents the downlink of data from the S/C

All of the following are examples of conditions that can be expressed with Atomic Constraints:

- `batterySOC` below 30
- `distanceFromSun` less than 0.5
- `cameraStatus` is `on`
- `downlinkData` is occurring

Behavior Specified by Atomic Constraints

The evaluation of a constraint can be thought of as the answer to the following query: when in the schedule is the condition specified by the constraint satisfied? Depending on the object being referenced by the Atomic Constraint, there are different types of conditions that can be expressed.

States

A `State` has built in convenience methods that can be used to specify constraints. The following methods provide queries about a State's value. A well configured IDE can help with discovery (via autocompletion) of the different constraint query methods available.

- `whenLessThan(Double x)`
- `whenGreaterThan(Double x)`
- `whenLessThanOrEqualTo(Double x)`
- `whenGreaterThanOrEqualTo(Double x)`
- `whenEqualWithin(Double x, Double tolerance)`

Mission modelers can also specify their own condition using the method:

- `when(Predicate<Double> condition)`

For example, a mission modeler could create a constraint that, when evaluated, will provide the occurrences when the battery state of charge is below 30% with the following code:

```
Constraint socMin = batterySOC.whenLessThan(30);
```

Alternatively, an adapter could chose not to use the convenience methods and supply their own lambda:

```
Constraint socMin = batterySOC.when(x -> x < 30);
```

Activities

The `ActivityTypeStateFactory` provides the `ofType` method, which can be used to specify the activity type of interest and returns an `ActivityTypeState`. The `ActivityTypeState` provides the `whenActive()` method, which returns a `Constraint`. Consider an adaptation that has a instantiated the variable `activities` to be an `ActivityTypeStateFactory` and includes a `downlinkData` activity. This `ActivityTypeStateFactory` and `ActivityTypeState` can be used to specify an activity type and create a constraint referencing that type, respectively:

```
Constraint whenDownlinkingData = activities.ofType("downlinkData").whenActive();
```

This constraint, when evaluated, will provide data which represents the start time and end time of each activity instance.

The Evaluation of Atomic Constraints & Windows

A constraint is evaluated against the schedule that is produced by a simulation. The schedule describes the simulated effects of activities, and captures when activity instances started and ended and how the value represented by a state changes over time. When an Atomic Constraint is evaluated, it is evaluated over the entire duration of a schedule. In other words, the constraint will be evaluated from the beginning of the schedule to the end of the schedule. If at any point during the schedule the behavior described by the constraint occurs, the constraint has been violated.

The violation of a constraint is captured by a `Window`. A `Window` is a Merlin-SDK object that has a start and an end `Duration`. The evaluation of a constraint produces a list of disjoint (non-overlapping) windows, where each window represents the duration of a violation. The start and end times of a violation can be produced by adding the start and end durations with the start time of the schedule. A user can use the `Aerie Planning UI` to visualize these violation windows.

For example, consider a constraint representing `batterySOC` below 30, and a schedule that starts at time 0µs and ends at 100µs. In this hypothetical schedule, the `batterySOC` starts at 100, and is set to 25 at 75µs, and is then set to 35 at 80µs. The evaluation of this constraint will produce one violation, which the Merlin-SDK will represent as a Window with the start duration as 75µs and the end duration as 80µs.

Compound Constraints and Logical Operators

Atomic Constraints can be used as building blocks to create more complicated constraints, Compound Constraints. Consider the following description of behavior that can be represented by a Compound Constraint:

```
When the S/C is less than 0.5AU from the Sun, the camera status is on
```

This behavior describes the windows during which the S/C is less than 0.5AU from the Sun and the camera status is on. This is equivalent to the intersection of windows produced by Atomic Constraints `distanceFromSun.whenLessThan(0.5)` and `cameraStatus.when(x -> x.equals("on"))`. The `Constraint` interface in the Merlin-SDK provides methods that can be used to build Compound Constraints and perform the required operations on sets of windows. The following methods are provided in the `Constraint` interface:

- `and(Constraint other)`: returns a Constraint representing the intersection of windows
- `or(Constraint other)`: returns a Constraint representing the union of windows
- `minus(Constraint other)`: returns a Constraint representing the subtraction of a sets of windows

While there is a conceptual difference between Atomic Constraints and Compound Constraints, they are both implemented with the `Constraint` interface in the Merlin-SDK. For instance, the example discussed in this section can be implemented with the following code:

```
Constraint cameraAltitude = cameraStatus.when(x -> x.equals("on")).and(distanceFromSun.whenLessThan(0.5));
```

Compound Constraints can also be implemented using the `exists` method provided by the ActivityTypeState class. The `exists` method accepts a `Function<ActivityInstanceState, Constraint>`, or a method that accepts an `ActivityInstanceState` object and returns a `Constraint` objects. The `exists` method can be used to specify a certain condition for each instance of the specified activity type

(see the next section for examples).

Creating Constraints for the UI: The Merlin-SDK Violable Constraint

The `ViolableConstraint` object in the Merlin-SDK should be used to create constraints in an adaptation. The ViolableConstraint wraps constraints with data necessary for producing constraint violations and visualizing in the Aerie Planning UI. For example, in addition to defining a violable condition, the ViolableConstraint class includes the following data:

- constraint id: a user provided id
- violation message: a user provided message describing the violation if it occurs
- violation category: e.g. "severe", "moderate", etc.
- name: a user provided name (Note violable constraint names must be unique)

  A list of all `ViolableConstraints` should be included in the states file such that the Querier (seen here) can easily get all ViolableConstraints to determine constraint violations for a simulation.

Example of a ViolableConstraint:

```
ViolableConstraint socConstraint
  =  new ViolableConstraint(batterySOC.whenLessThan(30).and(cameraStatus.when(x -> x.equals("on"))))
     .withId("minSOCAndCameraOn")
     .withName("Min Battery SoC and Camera On")
     .withMessage("Battery capacity too low for imaging")
     .withCategory("severe");
```

Currently, this `exists` method *must* be used when creating constraints referencing an activity, in order to associate any potential constraint violations with an activity id. For example, the following constraint describes the windows during which a `downlinkData` activity occurred:

```
ViolableConstraint downlinkDataOccurring =
    new ViolableConstraint(activities
        .ofType(downlinkData.class)
        .exists(act -> act.whenActive()))
        .withId("downlinking")
        .withName("Downlinking of data occurring")
        .withMessage("Nothing wrong - just an example constraint")
        .withCategory("none");
```

A more realistic Compound Constraint involving an activity type may describe the following scenario: `When the camera status is on, a downlink activity should not occur.`

```
ViolableConstraint downlinkingWithCameraOn =
    new ViolableConstraint(activities
        .ofType(downlinkData.class)
        .exists(act -> Constraint.and(
            act.whenActive(),
            cameraStatus.when(m -> m.equals("on")))))
        .withId("cameraOnandDownlink")
        .withName("Downlinking with Camera On")
        .withMessage("Potential for lost images")
        .withCategory("moderate");
```

Adding ViolableConstraints to a Merlin-SDK Simulation

A mission modeler can fork the Aerie repo, and replace the ViolableConstraints in SampleMissionStates referenced by the `List<ViolableConstraint> violableConstraints` variable to incorporate their constraints in their simulation.

Ensuring Unique Constraint Names

In Aerie 0.4, it is necessary to have a unique name for each `ViolableConstraint`. This can easily be checked by adding the following code snippet under your `violableConstraints` list:

```
List<ViolableConstraint> violableConstraints = List.of( ... );
static {
  final var constraintNames = new java.util.HashSet<>();
  for (final var violableConstraint : violableConstraints) {
    if (!constraintNames.add(violableConstraint.name)) {
      throw new Error("More than one violable constraint with name \"" + violableConstraint.name + "\". Each name must be unique.");
    }
  }
}
```

Constraint Violations

Whereas a `ViolableConstraint` defines a condition that *may* be violated, a `ConstraintViolation` object is used to represent actual violations of a `ViolableConstraint` for a given simulation. At the end of the execution of a simulation, the results of the simulation will be used to evaluate the `ViolableConstraints`. A constraint is violated if the behavior it specifies occurs.

The results of an execution of a simulation will include a list of constraint violations. Constraint violations can be reviewed in the UI. A constraint violation is represented by a `ConstraintViolation` type in the Merlin-SDK, and has the following information:

- constraint name - a user provided name
- constraint id - a user provided id
- constraint violation message - a user provided message the should describe the violation that occurred
- a list of windows representing the times during which violations occurred - this list is a result of constraint evaluation

The Merlin-SDK `SimpleSimulator` will return `SimulationResults` after executing a simulation, which contain a list of `ConstraintViolation` objects. The information contained by the list of constraint violations is serialized into a JSON format, and this data is then sent to the Aerie Planning UI. The UI can be used to inspect the windows of violated constraints. The UI visualizes a constraint with a transparent red box superimposed on a timeline. For example, the image below displays when a constraint on the `batteryCapacity` state is being violated.

images/ConstraintUI.png

User Guide

Aerie 0.3 Users Guide

Overview

This document is a guide to how to make use of current Aerie capabilities as of Aerie 0.3 delivery. Aerie is a new software system to support activity planning, sequencing and spacecraft analysis needs of missions. Aerie is being developed by the MPSA element of Multi-mission Ground System and Services (MGSS), a subsystem of AMMOS (Advanced Multi -mission Operations System). This guide will be updated as new features are added.

Aerie is a collection of loosely coupled services that support activity planning and sequencing needs of missions with modelling, simulation, scheduling and rule validation capabilities. Aerie will replace legacy MGSS tools including but not limited to APGEN, SEQGEN, MPS Editor, MPS Server, Slinc II / CTS and ULSGEN. Aerie 0.3 provides mainly the following:

1. Merlin adaptation framework offering a subset of APGEN capabilities,
2. Merlin web GUI for activity planning,
3. Merlin command line interface for activity planning, and
4. Falcon smart sequence editor GUI.

Prerequisites

An adaptation is simply a software that models spacecraft behavior while performing a set of activities over a plan duration. Merlin adaptations can simulate a variety of states that are perturbed by executed activities, and governed by system models. Merlin plans describe a scheduled collection of activity instances with specified parameters. Documentation below guides users on how to upload an existing adaptation jar file, how to create plans with that adaptation, and how to edit and simulate those plans. For users to complete these steps, they should be able to build adaptations and have access to an Aerie installation. For details of how to create adaptations with Aerie refer to the Mission Modeler Guide. For information on how to install Aerie services refer to the Product Guide.

Merlin Activity Plans

Merlin Activity Plans

Aerie provides a "plan service" that manages a repository of plans. Access to this repository is mediated by a REST-based interface exposed by the service. The repository itself is currently implemented with MongoDB. The repository of plans may be queried for a list of all plans, and new plans may be added to the repository. Existing plans may be retrieved in full, replaced in full or in part, or deleted in full. The list of activities in a plan may be appended to (by creating a new activity) and retrieved in full. Individual activities in a plan may be retrieved in full, replaced in full or in part, and deleted in full.

Operations on plans are validated to ensure consistency with the adaptation-specific activity model with which they are associated. Stored plans shall contain activities whose parameter names and types are defined by the associated activity type.

Plan Data composition

The plan repository contains a set of JSON-style documents with the following top-level attributes:

- id: A database-specific unique key identifying the plan.
- adaptationId: A database-specific foreign key identifying the adaptation that the plan was built against.
- name: A user-meaningful string specifying the name of the plan.
- startTimestamp: A DOY-style timestamp describing the instant at which the plan is considered to be applicable (e.g. “2020-198T01:23:45”).
- endTimestamp: A DOY-style timestamp describing the instant at which the plan is considered no longer to be applicable.
- activityInstances: A list of activity instances which comprise the plan.A plan is conceptually composed of activity instances. Each activity instance has the following attributes:
- id: A database-specific foreign key identifying the type of activity this instance represents.
- type: A string denoting the name of the activity type of this activity instance.
- startTimestamp: A timestamp describing the instant at which this activity instance is considered to begin.
- parameters: A map from parameter name to parameter value specifying the fine-grained behavior of this activity instance. These parameters are specified as part of an activity instance, and may recursively contain further parameter data. Each parameter may be an arbitrary JSON document, but their structure should match that expected by the parameter definition in the associated adaptation.

A formal Plan file SIS will be provided in upcoming deliveries. For a sample plan file see Appendix.

Aerie Command Line Interface

The Aerie command line interface for planning allows manipulation of plans in batch using the commands listed below. In order to use the command line interface users have run Aerie on their localhost, where the end points are hardcoded in an Aerie config file. In the future, the CLI can be configured to point to any host machine to make requests against. At that point, aerie services can run anywhere, and the CLI will work from any other machine as long as it has the proper host configured. Example input/output JSON is given in the appendix for reference.

Installation

The CLI is contained in a JAR file that can be downloaded from Artifactory. To install it, download the tar file located here. Note that you may need to log in to Artifactory to reach the download link. Decompress the file to your desired location. This can be done manually by going to your desired location in a terminal and entering

```
tar -xzf ~/Downloads/aerie-develop.tar.gz
```

The JAR will now be located at `<installation-path>/merlin-cli/merlin-cli.jar`. The CLI can now be run by executing

```
java -jar <installation-path>/merlin-cli/merlin-cli.jar
```

but this is cumbersome, so we suggest setting up an alias to run the cli. For shells such as Bash and Zsh (the default shells for recent Macintosh computers), this can be done as such:

```
alias merlin-cli="java -jar <installation-path>/merlin-cli/merlin-cli.jar"
```

At this point the CLI can be run by entering `merlin-cli` at the command prompt, however if a new terminal is opened the alias will need to be set up again. To avoid this, add the alias to your rc file, so it will be initialized every time your terminal starts up.

Below is an example of how to install the merlin CLI to `~/opt` after downloading the .tar.gz file in zsh:

```
cd ~/opt
tar -xzf ~/Downloads/aerie-develop.tar.gz
alias merlin-cli="java -jar ~/opt/merlin-cli/merlin-cli.jar"
```

Known Issues

It is possible when running the JAR file you will see something like the following error:

```
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.UnsupportedClassVersionError: gov/nasa/jpl/ammos/mpsa/aerie/merlincli/MainCLI
has been compiled by a more recent version of the Java Runtime (class file version 55.0), this version of the Java
Runtime only recognizes class file versions up to 52.0
```

If this error occurs, the cause is likely due to a mismatch in the version of Java being run and the version the JAR was compiled with. The CLI uses Java 11, so Java 11 should be used to run the CLI. Your java version can be checked by entering `java -version` at the command prompt.

Usage

The CLI is currently designed to automatically connect to a local deployment of the Aerie services (see here). In the future, the CLI will be configurable to connect to remote services, but at this time only use with a local deployment is possible.

Create a single adaptation

Create an adaptation from a JAR file and metadata, returns an adaptation ID.

```
merlin-cli --create-adaptation <path-to-adaptation-jar> name=<name> version=<version> mission=<mission> owner=<owner>
```

Query all adaptations

Returns a list of all adaptations indexed by adaptation ID.

```
merlin-cli --list-adaptations
```

Query adaptation metadata

Returns metadata of an adaptation specified with ID.

```
merlin-cli -a <adaptation-id> --view-adaptation
```

Query all activity types in an adaptation

Returns a list of activity types indexed by activity type name. For each activity type, parameter names and types as well as default values are given.

```
merlin-cli -a <adaptation-id> --activity-types
```

Query an activity type in an adaptation

Returns a list of parameters as well as their default values for an activity type specified by ID.

```
merlin-cli -a <adaptation-id> --activity-type <activity-type-id>
```

Query activity type parameters

Returns a list of activity type parameters for a given activity type within an adaptation, both specified by ID.

```
merlin-cli -a <adaptation-id> --activity-type-parameters <activity-type-id>
```

Delete an adaptation

Remove an adaptation specified by ID from the adaptation service.

```
merlin-cli -a <adaptation-id> --delete-adaptation
```

Create a plan

Create a plan by uploading a plan JSON file. The JSON should contain all required fields including adaptation ID, plan name, start timestamp and end timestamp (YYYY-DDDThh:mm:ss). Adaptation ID must be for a valid adaptation, and activity instances must be valid according to the adaptation.

```
merlin-cli --create-plan <path-to-plan-json>
```

Query plans

Returns a list of existing plans indexed by plan ID.

```
merlin-cli --list-plans
```

Query an activity instance

Given a plan and activity instance ID, return activity instance metadata and parameter list.

```
merlin-cli -p <plan-id> --display-activity <activity-instance-id>
```

Update plan metadata

Update plan metadata via key/value pairs. Valid fields are: `adaptationId`, `startTimestamp`, `endTimestamp`, `name`.

```
merlin-cli -p <plan-id> --update-plan <field>=<value>
```

Update an activity instance

Update any attribute of an activity instance specified by unique ID. Note that activity instance parameters cannot be updated this way. Valid fields are: `startTimestamp` and `name`.

```
merlin-cli -p <plan-id> --update-activity <field>=<value>
```

Delete an activity instance

Delete an activity instance from a plan by ID.

```
merlin-cli -p <plan-id> --delete-activity <activity-instance-id>
```

Delete a plan

Delete a plan specified by ID.

```
merlin-cli -p <plan-id> --delete-plan
```

Update plan from a file

Upload a JSON file containing fields of a plan to be updated. All fields except adaptation ID may be updated this way. Any fields that are left out will remain unchanged. Note that individual activity instances cannot be updated this way, and including any activity instances in the plan update JSON will replace the current activity instance set with them.

```
merlin-cli -p <plan-id> --update-plan-from-file <path-to-json>
```

Append activity instances to a plan by file upload

This action will retain activity instances in the plan, but add all new activity instances specified by a JSON file.

```
merlin-cli -p <plan-id> --append-activities <path-to-json>
```

Download a plan JSON

Download plan in a JSON file format. Note that a downloaded plan JSON is different from the format for creating a plan in that activity instances are given as a map indexed by activity ID instead of a list.

```
merlin-cli -p <plan-id> --download-plan <output-path>
```

Simulate a plan

Kick off a simulation of a plan by ID, with results written to a file. Takes a sampling period (in microseconds) to determine how often to sample states for simulation results. Note that the local version of this command currently ignores the output path, and instead prints results to the console.

```
merlin-cli -p <plan-id> --simulate <sampling-period> <output-path>
```

Convert an APGen APF file to a Merlin JSON Plan file

APF files can be converted to Merlin input JSON file format. The action requires specifying a path to the APGEN adaptation `.aaf` files.

```
merlin-cli --convert-apf <path-to-apf-file> <output-path> <path-to-apgen-adaptation-directory>
```

GraphQL SIS for Plan and Simulation Result

Aerie uses GraphQL as the API to interact with the system. One way to interact with Aerie is to use Apollo Client. Users can also use the playground to try out different queries. Note, some of the multations that require a unique input scalar (Upload) might not work using the playground.
Details on how to write a query and mutation can be foundhere.

Schema (as of Sep. 15th 2020)

1. Query Schema for Plan

```
type Query {
  ...
  plan(id: ID!): Plan
  plans: [Plan]!
  ...
}
```

Plan's Schema

```
type Plan {
  activityInstances: [ActivityInstance]!
  adaptation: Adaptation
  adaptationId: String!
  startTimestamp: String!
  endTimestamp: String!
  id: ID!
  name: String!
}
```

Activity Instance's Schema

```
type ActivityInstance {
  id: ID!
  parameters: [ActivityInstanceParameter]!
  startTimestamp: String!
  type: String!
}
type ActivityInstanceParameter {
  name: String!
  value: ActivityInstanceParameterValue!
}
```

Adaptation's Schema

```
type Adaptation {
  activityType(name: String!): ActivityType
  activityTypes: [ActivityType]
  id: ID!
  mission: String!
  name: String!
  owner: String!
  version: String!
}
```

Activity type's Schema

```
type ActivityType {
  name: String!
  parameter(name: String!): ActivityTypeParameter
  parameters: [ActivityTypeParameter]!
}
type ActivityTypeParameter {
  default: ActivityTypeParameterDefault
  name: String!
  schema: ActivityTypeParameterSchema
}
```

2. Query Schema for Simulation

```
type  Query {
  simulate(planId: String!, samplingPeriod: Float!): SimulationResponse
}
```

Simulation Response & Result's Schema

```
type SimulationResponse {
  message: String
  results: [SimulationResult!]
  success: Boolean!
  violations: [Violation!]
}
type SimulationResult {
  name: String!
  start: String!
  values: [SimulationResultValue!]!
}
type SimulationResultValue {
  x: Float!
  y: SimulationResultValueY!
}
```

3. Mutation Schema

Mutation schema for creating an activity instances

```
type Mutation{
  createActivityInstances(
    activityInstances: [CreateActivityInstance]!
    planId: ID!
  ): CreateActivityInstancesResponse
  ...
}
```

Mutation schema for creating an adaptation

```
type Mutation{
  ...
  createAdaptation(
    file: Upload!
    mission: String!
    name: String!
    owner: String!
    version: String!
  ): CreateAdaptationResponse
}
```

Mutation schema for creating a plan

```
type Mutation{
  ...
  createPlan(
    adaptationId: String!
    endTimestamp: String!
    name: String!
    startTimestamp: String!
  ): CreatePlanResponse
}
```

Mutation schema for update an activity instance

```
type Mutation{
  ...
  updateActivityInstance(
    activityInstance: UpdateActivityInstance!
    planId: ID!
  ): Response
}
```

Mutation schema for deleting an activity instance, adaptation and plan

```
type Mutation {
  ...
  deleteActivityInstance(planId: ID!, activityInstanceId: ID!): Response
  deleteAdaptation(id: ID!): Response
  deletePlan(id: ID!): Response
}
```

Usage

When writing a GraphQL query, please refer to the schema section for all valid fields that you can use for a particular query.

Query all plans

Returns metadata for all Plans

```
query {
  plans{
    id
    name
    startTimestamp
    endTimestamp
    adaptationId
  }
}
```

Query a single plan

Using the id you got from "Query all plans" to obtain information for a single plan.

Returns metadata of a single Plan

```
query {
  plan(id: "5f492c60ae0dec17320e5bed") {
    id
    name
    startTimestamp
    endTimestamp
    adaptationId
  }
}
```

Query all activity instances from a plan

You can either use "query plan" for all activity instances from a single plan or use "query plans" for all activity instances from all the plans

Returns activity instances metadata and parameter list

```
query {
  plan(id: "5f492c60ae0dec17320e5bed") {
    activityInstances{
      id
      startTimestamp
      parameters {
        name
        value
      }
    }
  }
}
```

Query adaptation from a plan

Returns metadata of an adaptation from a particular plan

```
query {
  plan(id: "5f492c60ae0dec17320e5bed") {
    adaptation {
      id
      mission
      name
      owner
      version
    }
  }
}
```

Query activity types within an adaptation from a plan

Returns a list of activity types. For each activity type, name and parameter list are given

```
query {
  plan(id: "5f492c60ae0dec17320e5bed") {
    adaptation {
      activityTypes{
        name
        parameters{
          name
          schema
        }
      }
    }
  }
}
```

Run simulation and query the result

You have to provide the planId and the simpling rate in order to run the simulation.

Returns a list of states with different sampling time.

```
query {
  simulate (planId: "5f492c60ae0dec17320e5bed", samplingPeriod: 10000000000 ) {
    message
    success
    results {
      name
      start
      values{
        x
        y
      }
    }
  }
}
```

Creating a activity instances

```
mutation {
  createActivityInstances(
     activityInstances:{parameters:{name:""}, startTimestamp:"2020-116T00:00:00", type: "TurnInstrumentOn"}, planId:"5f492c60ae0dec17320e5bed" )
  {
    message
    success
  }
}
```

Creating a adaptation

Note: The type "Upload" is a scalar in the schema. It is up to the server data source to interpret the file properly and forward the request to the services. References: MDN Web Docs and Apollo GraphQL Docs.

```
mutation {
  createAdaptation(file: <Upload>, mission: "EUROPA", name: "Test", owner: "Test", version: "0.1") {
    message
    success
  }
}
```

Creating a plan

```
mutation {
  createPlan(adaptationId: "5f492c3b8d1d733cf46f498e",
    startTimestamp:"2020-001T00:11:11.123",
    endTimestamp: "2020-001T11:11:11.123",
    name: "graphql",)
  {
    message
    success
  }
}
```

Deleting an activity instance

```
mutation {
  deleteActivityInstance(planId:"5f492c60ae0dec17320e5bed", activityInstanceId:"5f5d0601a88a0b4299b501c9")
  {
    message
    success
  }
}
```

Deleting an adaptation

```
mutation {
  deleteAdaptation(id:"5f492c3b8d1d733cf46f498e") {
    message
    success
  }
}
```

Deleting a plan

```
mutation {
  deletePlan(id:"5f5cef46a88a0b4299b501c7") {
    message
    success
  }
}
```

Updating an activity instance

```
mutation {
  updateActivityInstance(activityInstance: {
    id:"5f4e7c596d2c237b61fca4c6",
    parameters:{name:""},
    startTimestamp:"2020-116T00:00:00",
    type: "TurnInstrumentOff"},
    planId:"5f492c60ae0dec17320e5bed" )
  {
    success
    message
  }
}
```

Aerie Planning UI

Once Aerie services are installed on a mission server the Aerie planning web application will be available, which provides a graphical user interface to create, view, update and delete adaptations and plans. This section will refer to the demo instance of the UI available at: https://aerie-develop.jpl.nasa.gov

Uploading Adaptations

Adaptations can be uploaded to the adaptation service through the UI. To navigate to the adaptations page, click the `Adaptations` icon on the on the left navigation bar. Once an adaptation jar is prepared, it can be uploaded to the adaptation service with a name, version, mission and owner. The name and version must match (in case and form) the name and version specified in the adaptation.

For example, if the adaptation is defined in code as `@Adaptation(name="Bananaanation", version="0.0.1")`, then the name field must be entered as `Bananaanation` and the version as `0.0.1`. Once the adaptation is uploaded it will be listed in the table shown in Figure 1. Adaptations can be deleted from this table using the context menu by right clicking on the adaptation.

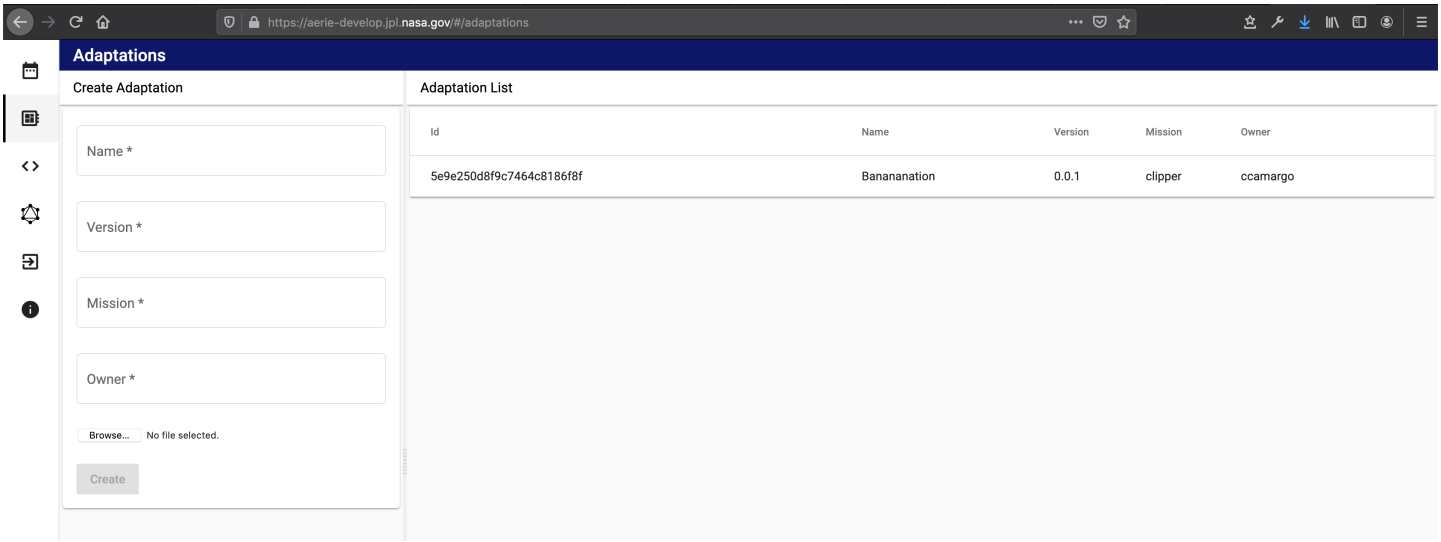


Figure 1: Upload adaptations, and view existing adaptations.

Creating Plans

To navigate to the plans page, click the `Plans` icon on the left navigation bar. Users can use the left panel to create new plans associated with any adaptation in the adaptation service. A `start` and `end` date has to be specified to create a plan. Existing plans are listed in the table on the right. Use right click on the table to reveal a drop down menu to delete and view plans.

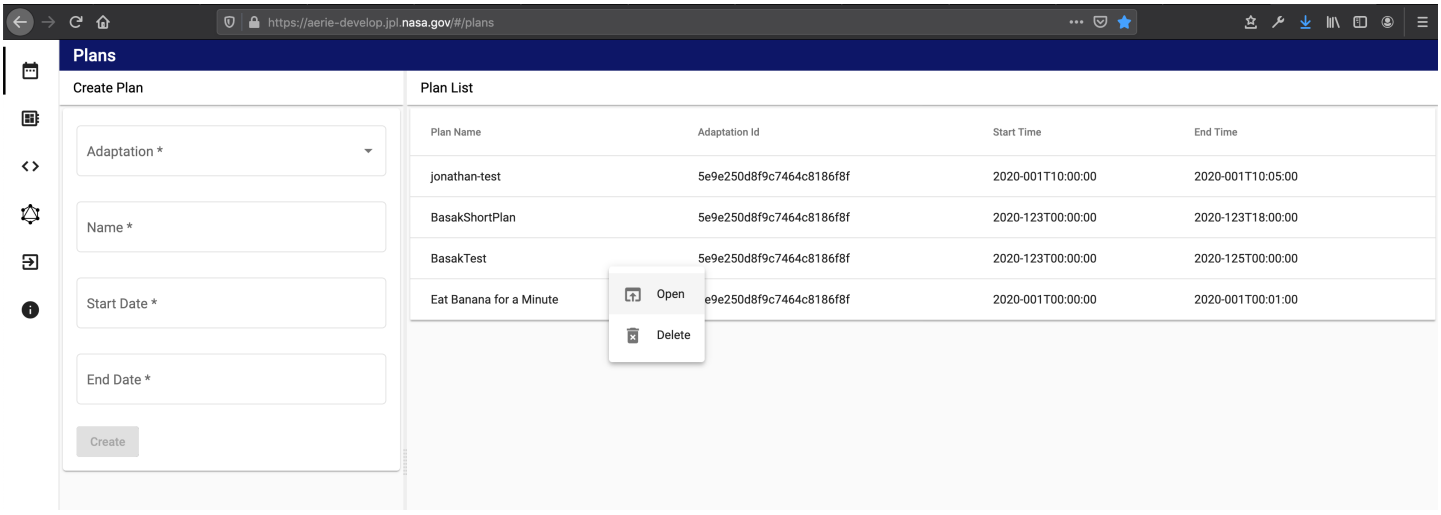


Figure 2: Create plans, and view existing plans.

View and Edit Plans

Once a user clicks on an existing plan, they can view contents, add/remove activity instances, and edit activity instance parameters. The plan view is split into the following default panels:

- Schedule Visualization
- Simulation Visualization
- Activity Instances Table
- Side drawer containing:
  - Activity Dictionary
  - Activity Instance Details

In the default side drawer the activity dictionary is displayed. Once a type or instance is selected, users can view details such as metadata and parameters by moving the arrow keys down. Activities can be dragged into the timeline from the activity dictionary. Once instances are added they will appear in the Activity Instances Table panel.

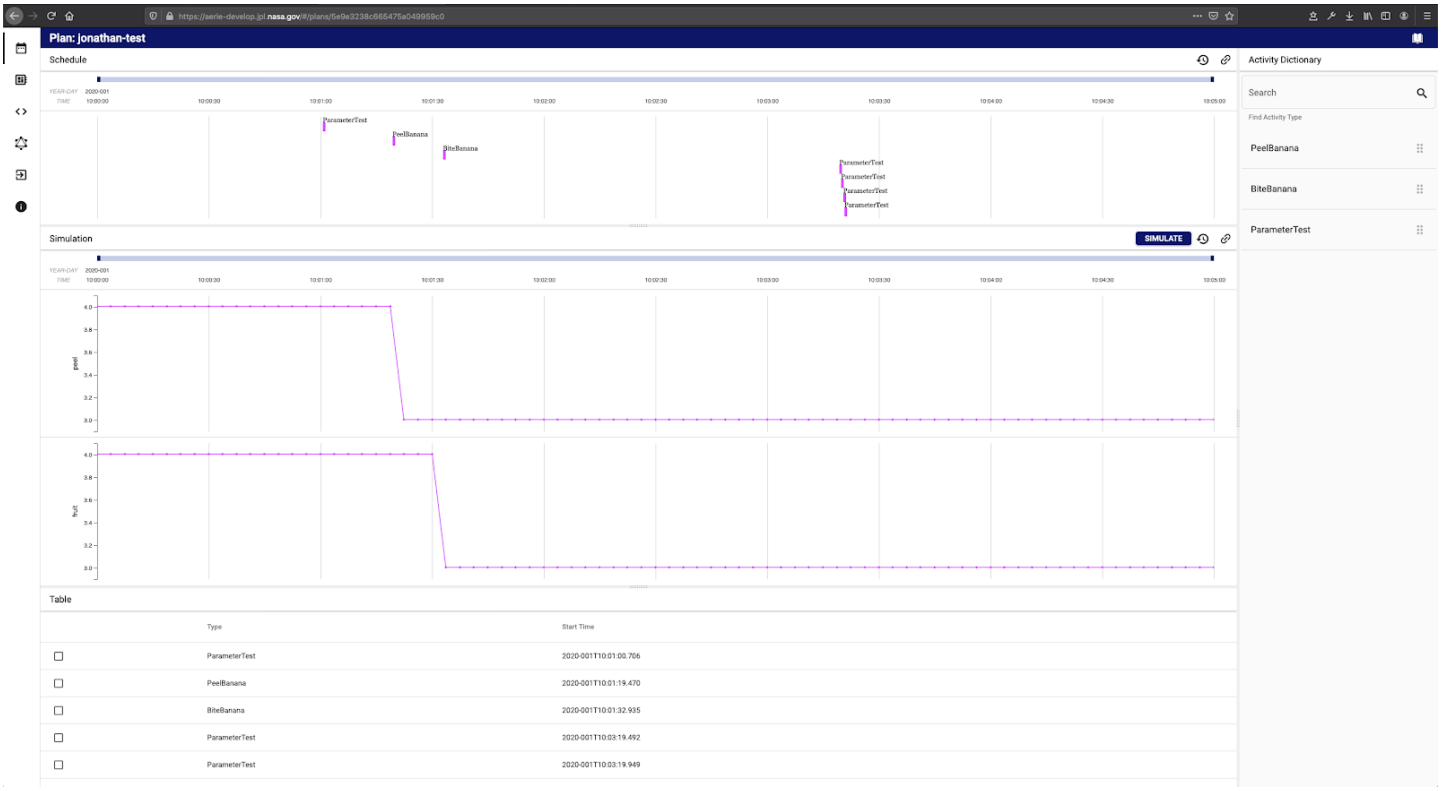


Figure 3: Default panels.

When a user clicks on an activity instance in the plan, the form to update activity parameters and start time will appear on the right drawer as shown in Figure 4. Users can use this form view to remove instances from the plan.

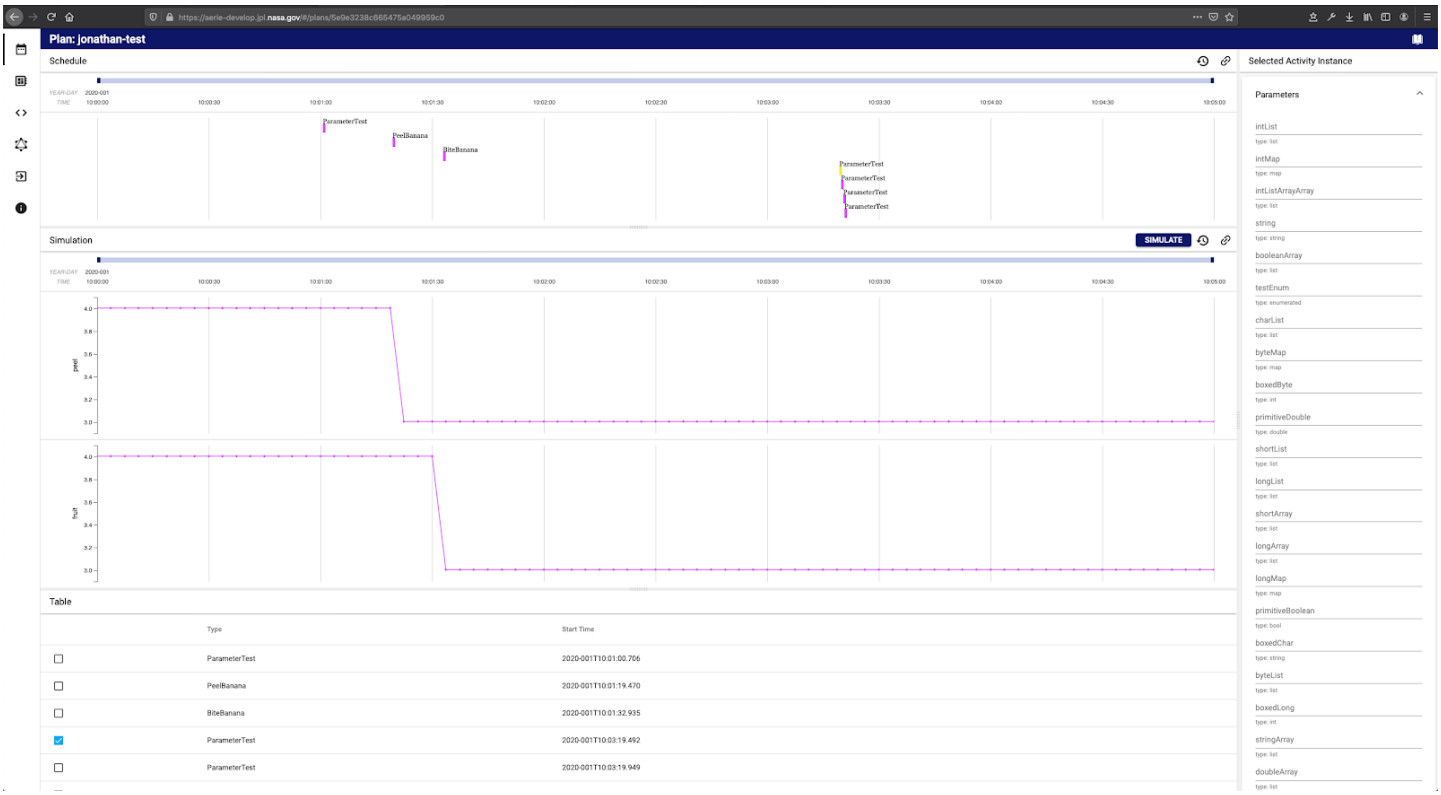


Figure 4: When an activity instance in plan is selected, its details will appear in the right drawer.

In the schedule and simulation elements, violations are shown as red regions in their respective bands as well as the corresponding time axis.

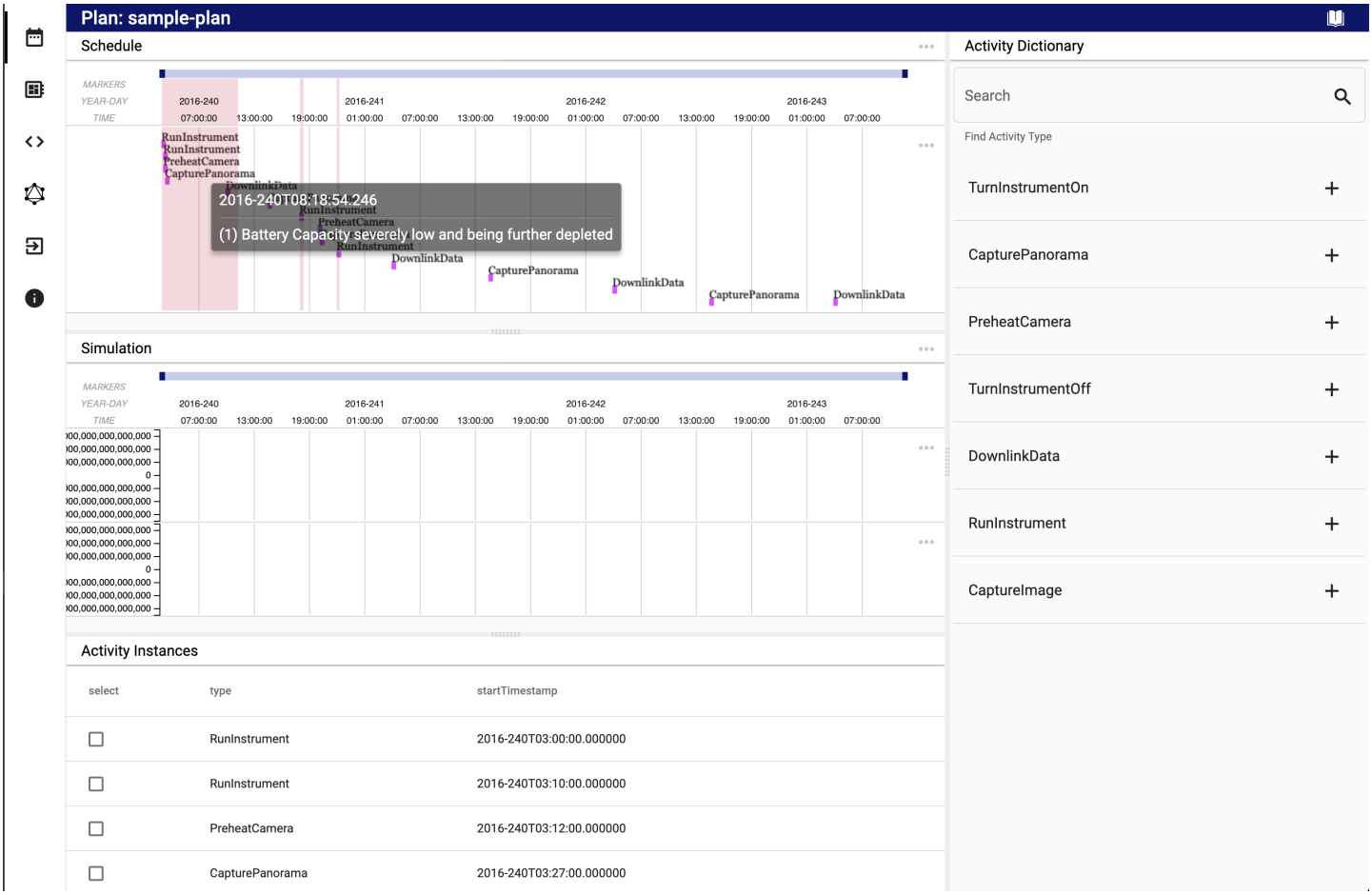


Figure 4.1: When violations occur, they will be represented as red areas on the timeline, with an accompanying hover state.

Horizontal guides may be added to any simulation or activity schedule band. The horizontal guides control UI may be accessed through each band's three dots more menu.

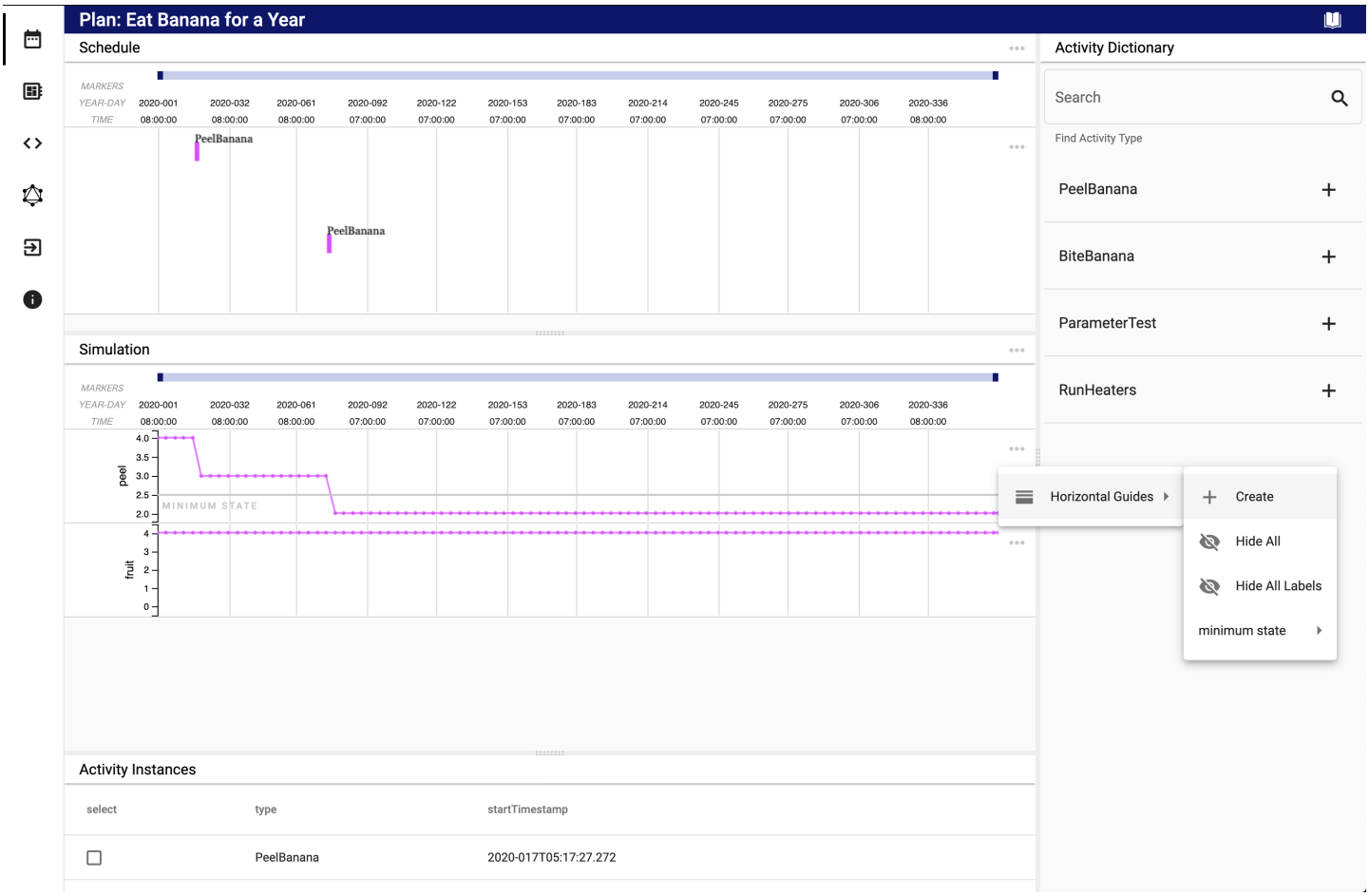


*Figure 4.2: Horizontal guides may be added to each activity schedule or simulation bands

Aerie UI provides a flexible arrangement where users can hide any of these panels by simply dragging dividers vertically. In Figure 5 this feature of the UI is illustrated.

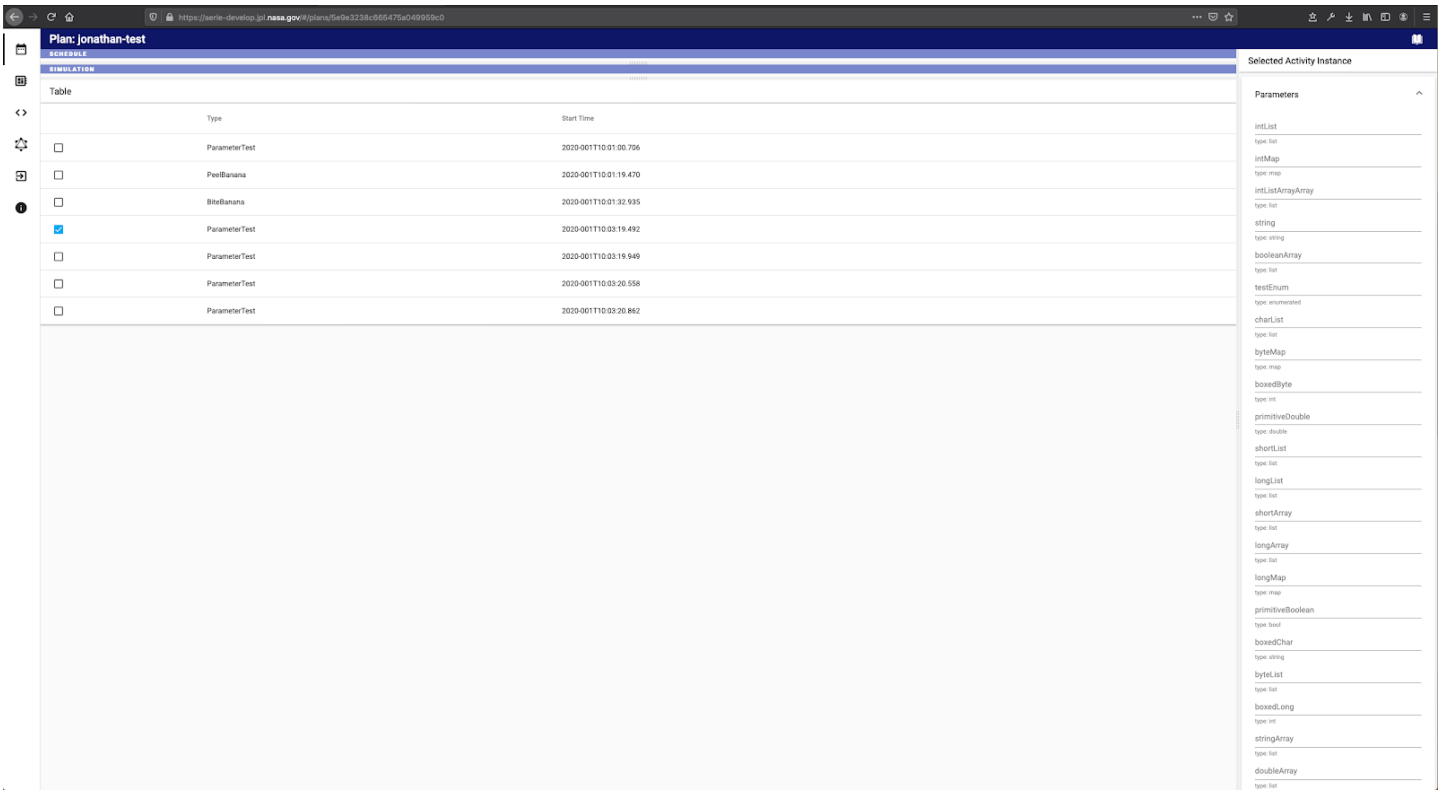


Figure 5: The bottom panels are dragged to the top edge completely leaving only one panels in view.

Note that all the default panels outlined here can be configured and changed based on the needs of a mission. You can read about how to do that in the UI Configurability documentation.

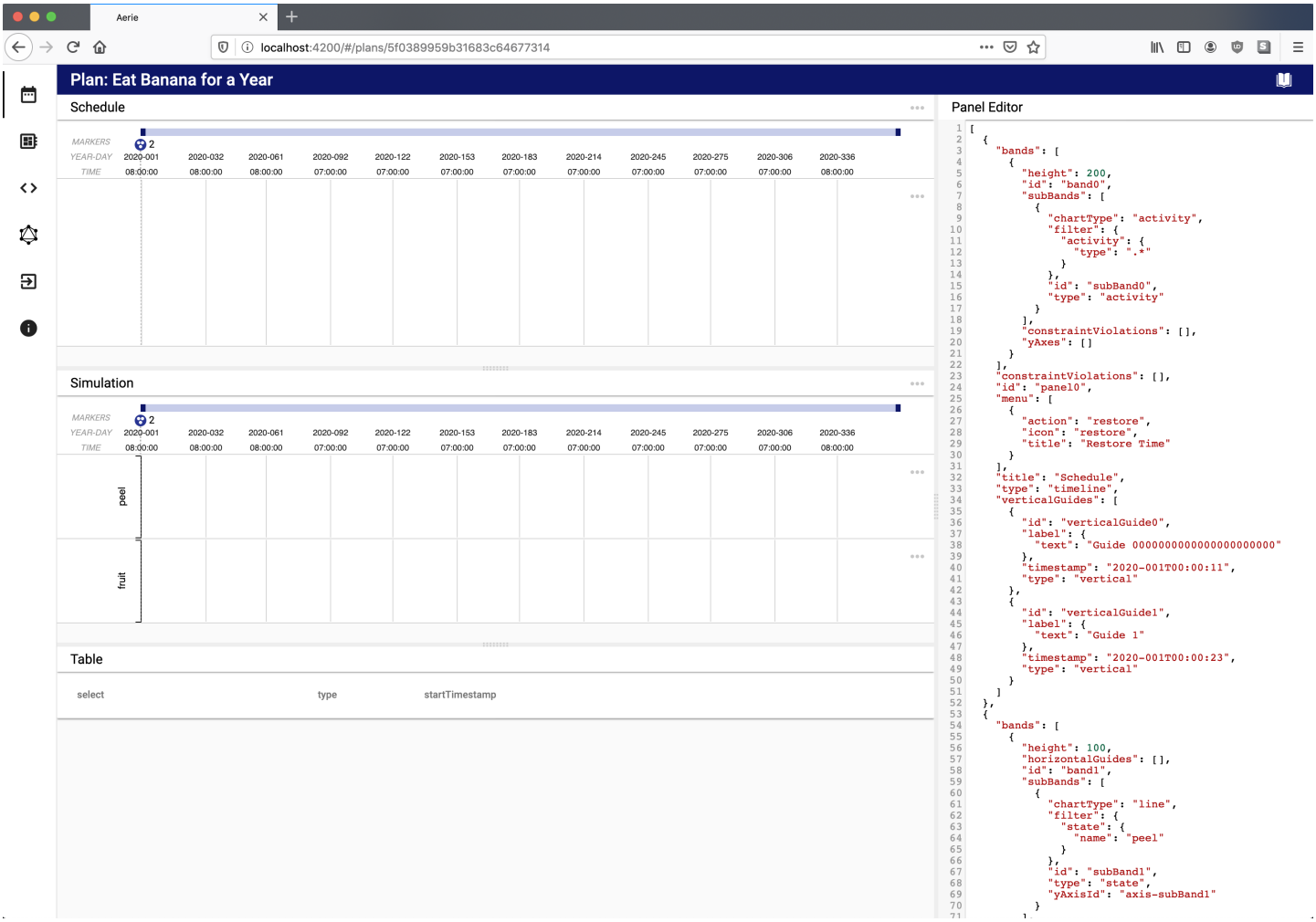
UI Configurability

Users can create custom planning views for different sub-systems
(e.g. science, engineering, thermal, etc.), where only data (e.g. activities and states) for those sub-systems are visualized. This is done through a mission-authored JSON file. The format of that file and how to update it is the subject of this document.

Panel Editor

In order to change the configuration object, navigate to the plan page
and press `⌘e (command e)`, which should open up the Panel Editor in the right drawer.
Make your changes to the JSON object and press `⌘s (command s)`.
You should see the UI update with your changes as long as your JSON object has no errors.
Check the console if you suspect your JSON object has errors. If your errors persist, don't hesitate to ask on#mpsa-aerie-users.

Note that edited configurations are not currently saved between browser refreshes. This will be added in future releases.



Panels

The planning UI consists of a list of panels, and each `Panel` has the following interface:

```
interface Panel {
  bands?: Band[]; // For type 'timeline'
  id: string;
  iframe?: { // For type 'iframe'
    src: string;
  };
  menu?: PanelMenuItem[];
  size: number; // Size percentage of the panel (0% - 100%)
  table?: { // For type 'table'
    columns: string[];
    type: 'activity'; // Only 'activity' typed tables are allowed right now
  };
  title: string;
  type: 'iframe' | 'table' | 'timeline';
}
```

Menu

Each panel can have an associated menu specified with an `action`, `icon`, and `title`.
The current actions we support are:

1. `link` adds a link that opens a new browser tab to the specified `url` in the `data` object.
2. `restore` is useful if the panel is typetimeline. It resets the timeline to it's max-time-rage (i.e. zooms all the way out).
3. `simulate` runs a simulation. Any state bands with simulation result states will be updated after a simulation.

More actions will be supported in the future and they will be more customizable. The icon should be a material icon. The menu interface looks like this:

```
export type PanelMenuItemAction = 'link' | 'restore' | 'simulate';

export interface PanelMenuItem {
  action: PanelMenuItemAction;
  data?: {
    url?: string;
  };
  icon: string;
  title: string;
}
```

A couple example menu JSON objects looks like this:

```
[
  {
    "action": "link",
    "data": {
      "url": "https://google.com"
    },
    "icon": "link",
    "title": "Google"
  },
  {
    "action": "restore",
    "icon": "restore",
    "title": "Restore Time"
  }
]
```

Types

There are currently three types of supported panels: iframe, table, and timeline. The following sections will detail how to create each of these panel types.

Inline Frame (iframe)

The iframe panel allows you to embed a custom HTML page inside of the panel. To create an iframe panel you need to specify `type: "iframe"`, a unique `id`, a `size`, a `title`, and an `iframe` object with a `src`. The `src` is a URL of the HTML page you are embedding in the panel. Here is a basic example of specifying an iframe panel:

```
{
  "id": "panel3",
  "iframe": {
    "src": "https://www.chartjs.org/samples/latest/charts/line/basic.html"
  },
  "size": 100,
  "title": "Line Chart",
  "type": "iframe"
}
```

Table

The table panel allows you to view data as a table with columns and rows. You can currently only create table panels for activity data. To create a table panel you need to specify `type: "table"`, a unique `id`, a `size`, a `title`, and a `table` object. The table object specifies the columns you want to see in your data. For example if you want to see an activities `startTimestamp` property you specify it in the column. There is also a special `select` column that allows the column to be selected. Here is a basic example of specifying a table panel:

```
{
  "id": "panel2",
  "size": 100,
  "table": {
    "columns": [
      "select",
      "type",
      "startTimestamp"
    ],
    "type": "activity"
  },
  "title": "Activity Table",
  "type": "table"
}
```

Timeline

The timeline panel allows you to specify visualizations of time-ordered data. To create a timeline panel you need to specify `type: "timeline"`, a unique `id`, a `size`, a `title`, and a list of `bands`. Here is the minimum JSON needed to specify a timeline panel:

```
{
  "bands": [],
  "id": "panel0",
  "size": 100,
  "title": "My First Panel",
  "type": "timeline"
}
```

To visualize data in a timeline you need to add band objects to the `bands` array. A band is a layered visualization of time-ordered data. Each layer of a band is specified as an object of the `subBands` array. The interfaces for a `Band` and `SubBand` are as follows:

```
interface Band {
  height?: number;
  id: string;
  subBands: SubBand[];
  yAxes?: Axis[];
}

interface SubBand {
  chartType: 'activity' | 'line' | 'x-range'; // How we actually visualize the data
  color?: string; // Color of all data points
  filter?: {
    activity?: {
      type?: string; // JS regex filtering the type of activity we want to see
    };
    state?: {
      name?: string; // JS regex filtering the name of state we want to see
    };
  };
  id: string;
  type: 'activity' | 'state'; // Type of data the sub-band visualizes
  yAxisId?: string; // Optional y-axis ID link
}
```

Here is a JSON object that creates a single band with one activity sub-band. Notice there are no `yAxes`, as activities do not

typically have y-values. Also notice the `filter` property, which is a JavaScript Regular Expression that specifies we only want to see `activity` of `type` `.*`. This is a regex for giving all activity types.

```
{
  "id": "band0",
  "subBands": [
    {
      "chartType": "activity",
      "filter": {
        "activity": {
          "type": ".*"
        }
      },
      "id": "subBand0",
      "type": "activity"
    }
  ]
}
```

For data that has y-values (for example state data), you can specify a y-axis and link a sub-band to it by ID. Here are the interfaces for `Axis` and `Label`:

```
interface Axis {
  id: string;
  color?: string;
  label?: Label;
  scaleDomain?: number[];
  tickCount?: number;
}

interface Label {
  align?: CanvasTextAlign;
  baseline?: CanvasTextBaseline;
  color?: string;
  fontFace?: string;
  fontSize?: number;
  hidden?: boolean;
  text: string;
}
```

Y-axes are specified in the band separately from sub-bands so we can specify multi-way relationships between axes and sub-bands. For example you could have many sub-bands corresponding to a single axis.

Here is the JSON for creating a band with two overlayed `state` sub-bands. The first sub-band shows only states with the name `peel`, and uses the y-axis with ID `yAxis1`. The second sub-band shows only states with the name `fruit`, and uses the y-axis with the ID `yAxis2`.

```
{
 "id": "band1",
 "subBands": [
  {
   "chartType": "line",
   "filter": {
    "state": {
     "name": "peel"
    }
   },
   "id": "subBand1",
   "type": "state",
   "yAxisId": "yAxis1"
  },
  {
   "chartType": "line",
   "filter": {
    "state": {
     "name": "fruit"
    }
   },
   "id": "subBand2",
   "type": "state",
   "yAxisId": "yAxis2"
  }
 ],
 "yAxes": [
  {
   "id": "yAxis1",
   "label": {
    "text": "peel"
   }
  },
  {
   "id": "yAxis2",
   "label": {
    "text": "fruit"
   }
  }
 ]
}
```

Aerie Editor (Falcon)

Aerie Editor (Falcon) is a web-based Visual Studio Code (VS Code) text editor that supports code completion, error diagnostics, tooltips, syntax highlighting, snippets, bracket-matching, auto-indentation, and many more robust features.

Installation

The editor needs to be installed on a machine for each user in their home directory. This ensures the users files are available for editing in the explorer and are not exposed to other users on the system. You can find more complete installation instructions here.

Basic Usage

The editor can be integrated with mission SEQGEN adaptations by adding an SMF file to the editor workspace. The editor parses SMF files in the workspace and uses them to validate sequences. The editor also outputs sequences in both the SASF and SATF formats, which can be ingested by SEQGEN to create an SSF file. In Figure 1 below, the SMF file can be seen in the workspace. As users start typing valid command stems, an autocomplete popover appears, filtering and listing all commands starting with that stem.

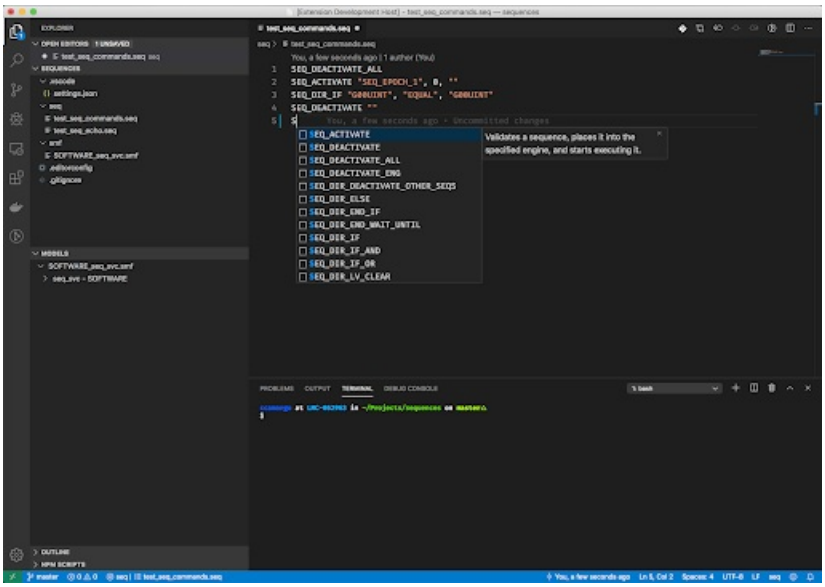


Figure 1: Code completion of commands based on SMF file in the workspace.

Additionally, Figure 2 shows the support for real-time sequence validation as users type commands. For wrong command stems, missing parameters, or wrong parameter names, the incorrect area will be underlined in red and an error message tooltip will be displayed upon hover.

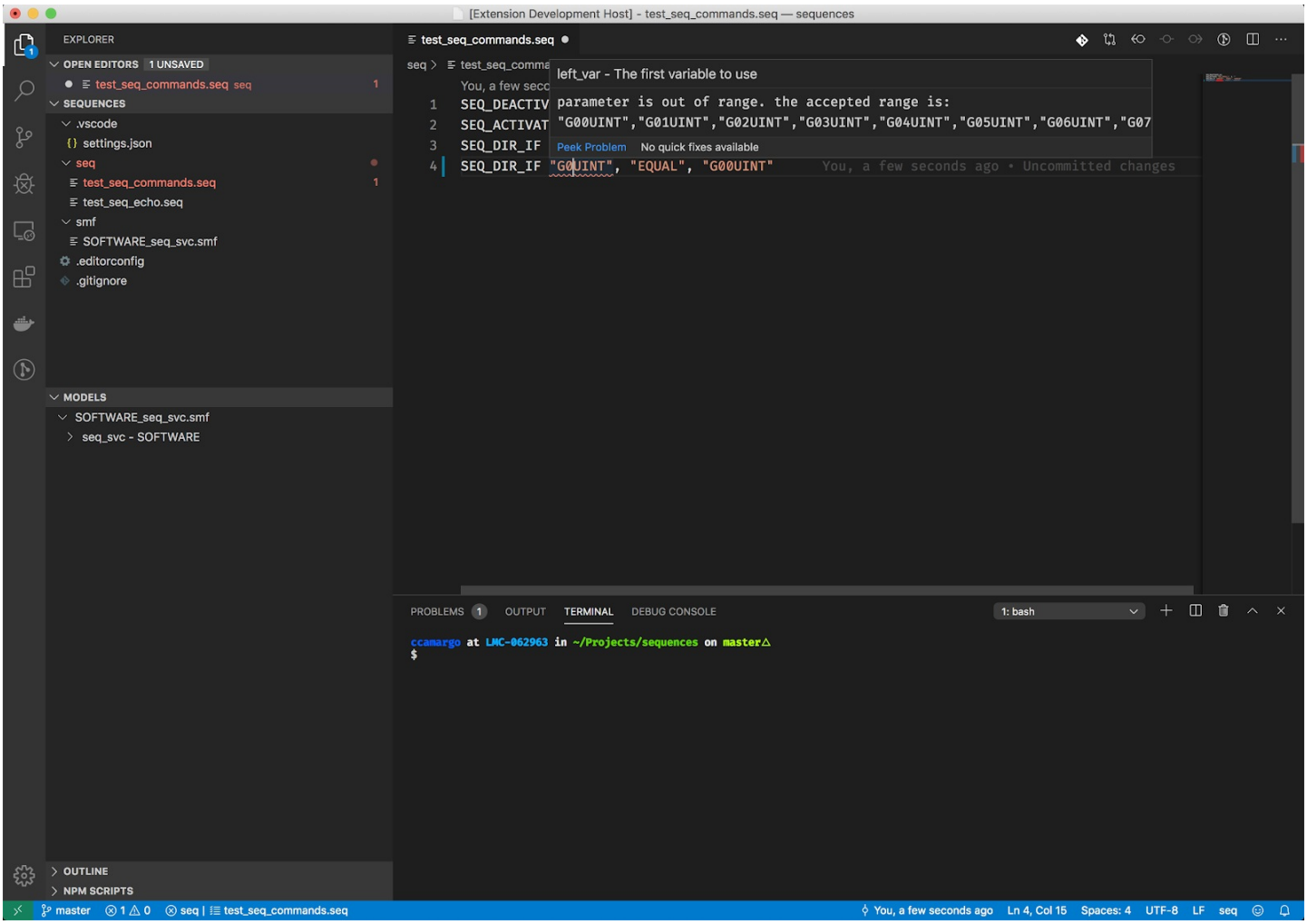


Figure 2: Error diagnostics of incorrect parameter names for a command.

Git Integration

All files in the workspace live on the editor installation machine. However, there is inherent support for pointing the application to a Git repository where files may be shared and stored for the whole mission. Git is available if the workspace is pointed to a directory that contains a .git directory. For more advanced integration on VS Code and Git, please refer to the VS Code Git documentation.

Explorer

Users can create, delete, and rename files and directories via the Explorer. For a more detailed explanation see the VS Code Explorer documentation.

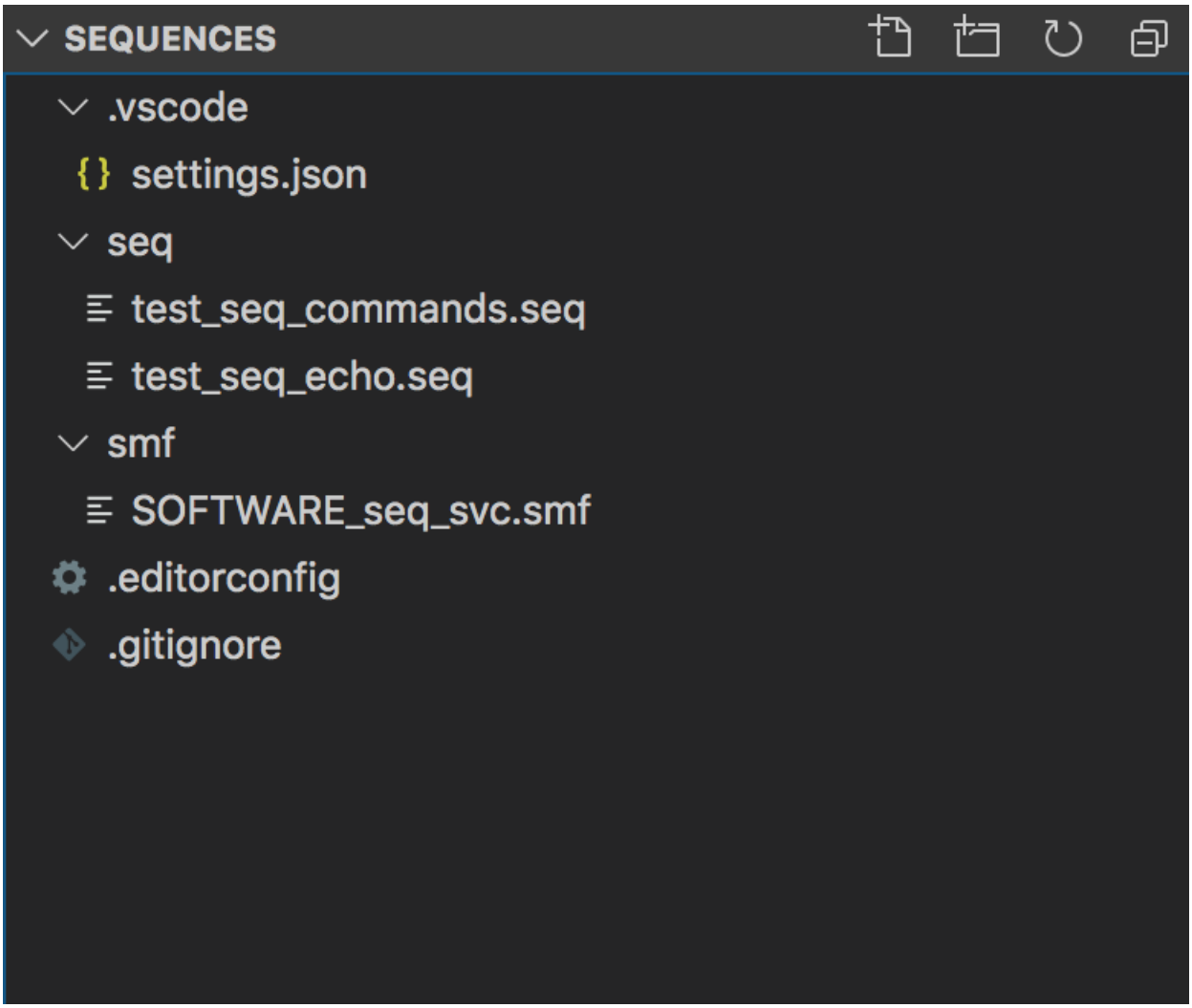


Figure 3: To create files and folders use the icons shown at the upper right corner of the Explorer.

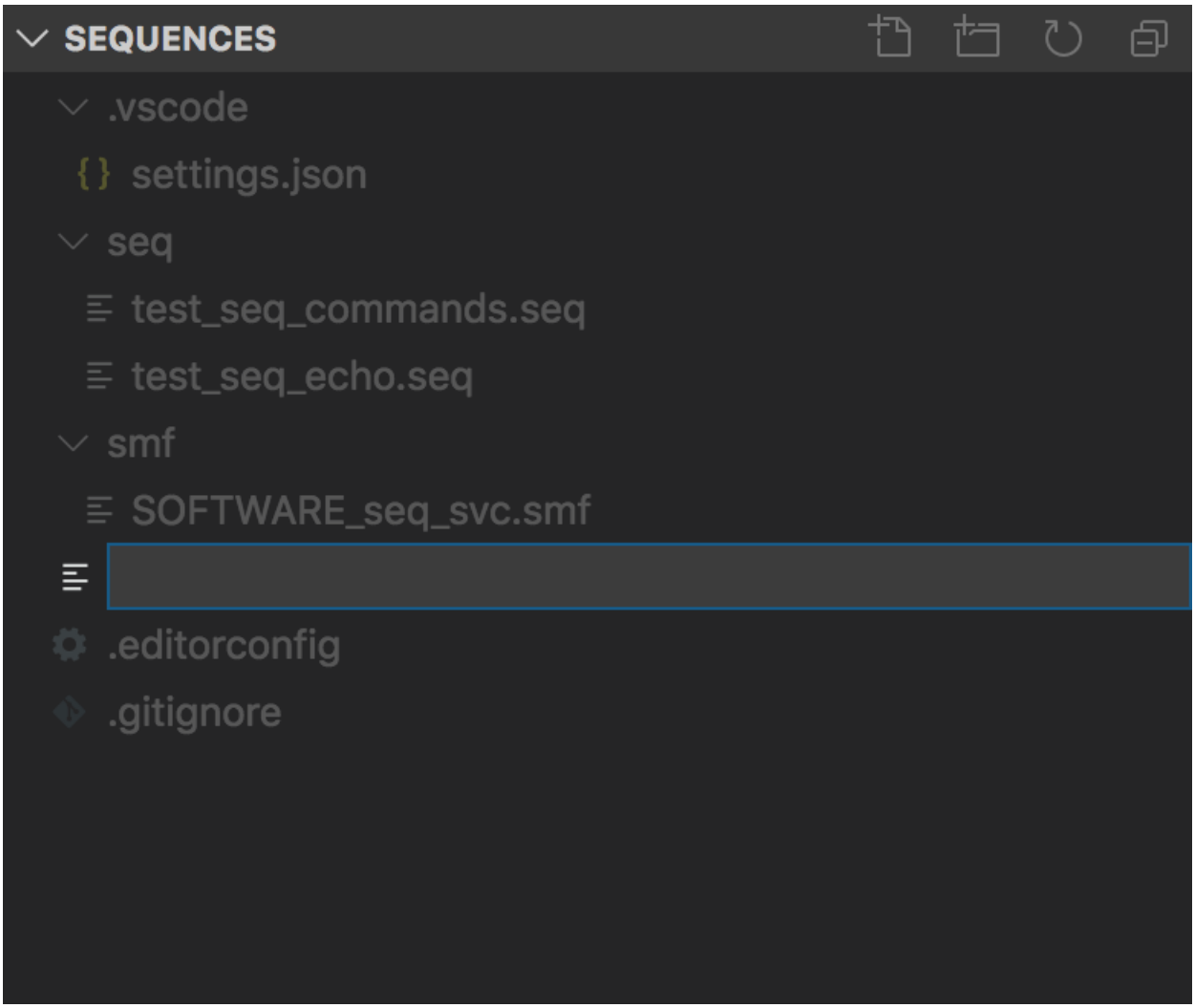


Figure 4: To rename any file or folder double click on the name, which will turn name field into an input box.

SEQGEN Integration

To invoke SEQGEN, click the button that shows up at the bottom of the editor window that say`run Seqgen`. Note that the button will only show up if SEQGEN and eurcseq is installed on the machine (on the `PATH`) that the editor is installed on. After running SEQGEN, the SSF file will be added to the editor workspace. If the workspace is a Git repository then the output files will be added to the repo as changed files but not as commits. To add them to the repo, users need to do a commit and push.

Configuration

To configure running SEQGEN, you need to create a file called `settings.json` in the .vscode folder in your workspace. It should have the following properties:

```
{
  "seq.seqgenInputDir": "",
  "seq.seqgenOutputDir": "",
  "seq.smfDirs": []
}
```

Property	Description
seqgenInputDir	Absolute path to the directory containing `.seq` files
seqgenOutputDir	Absolute path of output directory for SEQGEN
smfDirs	A list of directory absolute paths containing SMF files

User Guide Appendix

Appendix

CLI JSON DOWNLOAD PLAN FORMAT SAMPLE

```
{
   "name": "example_plan",
   "adaptationId": "5df16e65a920f467637bac3a",
   "startTimestamp": "2018-331T00:00:00",
   "endTimestamp": "2018-332T00:00:00",
   "activityInstances": {
      "5e1caea5176cfc2d58b2c54a": {
         "type": "BiteBanana",
         "startTimestamp": "2018-331T04:00:00",
         "parameters": {
            "biteSize": 7.0
         }
      },
      "5e1caea5176cfc2d58b2c54b": {
         "type": "PeelBanana",
         "startTimestamp": "2018-331T04:00:00",
         "parameters": {
            "peelDirection": "fromStem"
         }
      }
   }
}
```

CLI JSON UPLOAD PLAN FORMAT SAMPLE (No unique ID for activity instances)

```
{
   "adaptationId": "5df16e65a920f467637bac3a",
   "endTimestamp": "2018-331T00:00:00",
   "name": "example_plan",
   "startTimestamp": "2018-332T00:00:00",
   "activityInstances": [
    {
         "type": "BiteBanana",
         "parameters": {
               "biteSize": 7.0
         },
         "startTimestamp": "2018-331T04:00:00"
    },
     {
         "type": "PeelBanana",
         "parameters": {
            "peelDirection": "fromStem"
         },
        "startTimestamp": "2018-331T04:00:00"
    }
   ]
}
```

CLI JSON APPEND ACTIVITY INSTANCES FORMAT SAMPLE

```
[
   {
      "type": "BiteBanana",
      "parameters": {
            "biteSize": 7.0
      },
      "startTimestamp": "2018-331T04:00:00"
   },
   {
      "type": "PeelBanana",
      "parameters": {
         "peelDirection": "fromStem"
      },
      "startTimestamp": "2018-331T04:00:00"
   }
]
```

QUERY AN ACTIVITY TYPE FOR AN ADAPTATION OUTPUT SAMPLE

```
{
  "parameters": {
    "peelDirection": {
      "type": "string"
    }
  },
  "defaults": {
    "peelDirection": "fromStem"
  }
}
```

Product Guide

Product Installation

Installation Instructions

Installation instructions are found in the Aerie repository deployment documentation. If you have any questions or issues, don't hesitate to ask on #mpsa-aerie-users.

Docker Containers

Goto the Artifactory Aerie Docker repository and log in with your JPL credentials. The latest released containers are:

```
docker-release-local/gov/nasa/jpl/aerie/adaptation/release-0.4.0
docker-release-local/gov/nasa/jpl/aerie/plan/release-0.4.0
docker-release-local/gov/nasa/jpl/aerie-apollo/release-0.4.0
docker-release-local/gov/nasa/jpl/aerie-ui/release-0.4.0
```

Example Docker-Compose

An example Docker Compose file is available for deployment. You can use these instructions to help you deploy.

```
## TARs

If you just want the Aerie JAR files you can find them at:
```

general/gov/nasa/jpl/aerie/aerie-release-0.4.0.tar.gz

```
The Aerie Editor (Falcon) can be found at:
```

general/gov/nasa/jpl/aerie/aerie-editor-release-0.4.0.tar.gz `

Known Issues

1. When using the IntelliJ IDE, upon a source file change, only the affected source files will be recompiled. This causes conflicts with the annotations processing being used for Activity Mapping. For now manually rebuilding every time is the solution.

System Requirements

Software Requirements

Name	Version
DOCKER	19.X
*NODEJS	12.X LTS
*NPM	6.X
*OPEN JDK	11.X

*For build purposes only. Not needed for installing the application.

Supported Browsers

Name	Version
CHROME	LATEST

Name	Version
FIREFOX	LATEST

Hardware Requirements

Hardware	Details
CPU	2 GIGAHERTZ (GHZ) FREQUENCY OR ABOVE
RAM	4 GB AT MINIMUM
DISPLAY RESOLUTION	2560-BY-1600, RECOMMENDED
INTERNET CONNECTION	HIGH-SPEED CONNECTION, AT LEAST 10MBPS

TCP Port Requirements

Service	Port
Aerie UI	8080
Adapataion	27182
Plan	27183
RabittMQ	15762
Aerie Apollo	27184

Administration

This product is using Docker containers to run the application. There are total of five Docker containers that are internally bridged (connected) to run the application. Containers can be restarted in case of any issues using Docker CLI. Only port 8080 from the UI container is exposed to outside.

Environment Variables

Aerie software does not have any environment variables at this point in time.

Network Communications

The Aerie deployment configures the port numbers for each container via docker-compose. The port numbers must match those declared within the services' config.json. In a large majority of Aerie deployments no change to these port numbers will be needed, nor should one be made. The only port number that might be desired to change is the Aerie-UI port (8080). in this case the number to change is the first port number of the pair [XXXX:XXXX]. The second number represents the port number within the container itself. An example of this would be ports: ["8080:80"]. The number that needs to be changed is the first port which is 8080.

Environment Variables

Port Type	Default Port Number	Description
TCP	8080	UI Port

Administration Procedures

Aerie is orchestrated as a set of Docker containers. Each of the software components are packaged and run in an isolated docker container independently from one another. There exists seven docker containers:

- Aerie-UI: Hosts the web application and communicates with Aerie via the GraphQL Apollo Server.
- Aerie-Apollo: The GraphQL Apollo Server which functions as the Aerie API Gateway against which clients can submit GraphQL queries and mutations.
- Adaptation: Handles all the logic and functionality for the model Adaptation for activity planning.
- Adaptation-mongo: Holds the data for Adaptation container.
- Plan: Handles all the logic and functionality for activity planning.
- Plan-mongo: Holds the data for Plan container.
- RabbitMQ: The Aerie services' message bus message exchange instance.

The Adaptation, Plan, and Apollo servers communicate to each other via a REST API (internal to Aerie) with the ports specified in the docker-compose file. The database containers, Adaptation-mongo and Plan-mongo are isolated to connect only with their respective service container (Adaptation or Plan).

Product Support

Defect Reporting Procedure

All defect reports should go toaerie_support@jpl.nasa.gov.

Points of Contact

- Adaptation: Kenneally, Patrick W, Development Lead
- Administration: Kenneally, Patrick W, Development Lead
- General Help: Alper Ramaswamy, Emine Basak, Product Lead
- #mpsa-aerie-users: User help Slack channel