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 our `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(
    public static final IndependentStateFactory factory = new IndependentStateFactory(query,

    public static final SettableState<Boolean> heaterPoweredOn = factory.bool("heaterPowered
    public static final DoubleState batteryCapacity = factory.cumulative("batteryCapacity",

    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 construct activity 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", 1000
```

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",
```

Integer value

```
public static final SettableState<Integer> samplesAcquired = factory.integer("samplesAcquire
```

Double value

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

Boolean value

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

Enumerated value

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

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, `ParameterMapper<T> mapper`. This method allows adaptation engineers to provide a `ParameterMapper` (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 `CustomTypeParameterMapper`:

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