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>
        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:

```java
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>> reactionCon
    private final static DynamicCell<ExampleQuerier<?>.StateQuerier> stateContext = DynamicC

    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 DynamicReactior

    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>>> se
    private final Map<String, Query<T, ExampleEvent, RegisterState<Double>>> cumulables = ne

    public ExampleQuerier(final ActivityMapper activityMapper, final SimulationTimeline<T, E
```

```
        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
            this.stateNames.add(name);

            final var query = timeline.register(
                    new SettableEffectEvaluator(name).filterContramap(ExampleEvent::asIndepe
                    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
            this.stateNames.add(name);

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

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

    @Override
    public void runActivity(ReactionContext<T, Activity, ExampleEvent> ctx, String activityI
        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> hi
    if (this.settables.containsKey(name)) return this.settables.get(name).getAt(history)
    else if (this.cumulables.containsKey(name)) return SerializedParameter.of(this.cumul
    else throw new RuntimeException("State \"" + name + "\" is not defined");
}

@Override
public List<ConstraintViolation> getConstraintViolationsAt(History<T, ExampleEvent> hist
    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
        final var violationWindows = stateContext.setWithin(stateQuerier, violableConstr
        if (!violationWindows.isEmpty()) {
            violations.add(new ConstraintViolation(violationWindows, violableConstraint)
        }
    }

    return violations;
}

public StateQuery<SerializedParameter> getRegisterQueryAt(final String name, final Histo
    if (this.settables.containsKey(name)) return this.settables.get(name).getAt(history)
    else if (this.cumulables.containsKey(name)) return StateQuery.from(this.cumulables.g
    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.