- 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 (see here) 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.

```java
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 PowerOffHe
        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.