ExploDTwin documentation

ExploDTwin team

March 8, 2026

Contents

1 ExploDTwin library

The ExploDTwin library is a SysML v2 library. It provides definitions the Digital Twin designer must use to implement a formal model of a system. This library is part of a DT framework, where the pipeline is roughly the following:

1. The DT designer provides a compliant SysML v2 model.
2. The What-If Analysis component of the framework parses the model and creates a plan according to some user-provided constraints and goals. The plan is then sent to the Physical Twin for execution.
3. The Fault Detection, Isolation and Recovery component also parses the model and creates a monitor of the plan, provided by the WIA.
4. While performing the plan, the PT produces telemetries, that are consumed by the FDIR component.

Parts of the library are tailored with the above components in mind. The library file structure is the following.

- `ExploDtwinMetamodel.sysml`. The user-implemented model is supposed to adhere to a predefined structure, described by the metamodel, which we provide here.
- `ExploDTwinMetadata.sysml`. Data not strictly related to the model (e.g. versions, descriptions, and more) can be embedded via metadata annotations. Metadata is useful for the DT framework components that process the model.
- `ExploDTwinTypes.sysml`. The library implements a number of types that are not offered by SysML v2's standard library.

1.1 ExploDTwin Metamodel

The ExploDTwin library contains the components the DT designer must use to implement a model of the PT. The components and a brief description follows.

- `Subsystem`. A subsystem is a component or module of the PT. Each subsystem is related to a specific concern, or task.
- `Activity`. Subsystems perform activities. Activities change the state of the subsystem.
- `RealAsset`. The real asset is a collection of subsystems, it is the model of the PT. It describes the relationship between the subsystems.
- `Environment`. The environment in which the PT operates.
- `SystemComponent`. Contains the whole DT, and describes the relationship between environment and real asset.

1.1.1 Activity

An activity can be either an atomic movement, or action, performed by a single subsystem.

In the library, `Activity`'s semantics is provided via a state machine, which initially stays idle, then progresses and finally returns in its idle state.

The behavior of an activity is hidden to the DT designer. The DT designer defines an activity, its constraints and effects. The connection between subsystems and activities is realized by durative transitions, futher described in paragraph 1.1.2.

SysML implementation An activity can start only if its pre-conditions are satisfied, and if it stops, it means its post-conditions are met. Activities have input-parameters, pre-effects, post-effects and an invariant. Pre- and post-effects are assignments to that are executed at the start, and after the end of the activity, respectively. While pre, post-condition and invariant are boolean expressions that must be true before the start, during and after the end of the activity.

To create a new activity, the DT designer redefines the abstract parameters. The DT designer also lists all the necessary inputs, and whether they are input parameters of the activity or values that should be acquired externally (for example from a subsystem). These inputs are the variables that are then used to define conditios and effects.

Activities have parameters, but they might need other inputs, for example, they might need to know variables of the subsystems they interact with. We provide the metadata definition `ActivityParam`. Any attribute defined inside an activity, labelled by such metadata, will be considered an activity parameter.

1.1.2 Subsystems

A subsystem can be modelled as a state machine whose state can change because of either an activity, a guard, or time.

Each subsystem has an interface, which is specified via SysML attributes with either one of `in` and `out` keywords: the value of the attribute is an input of the subsystem, or an output, respectively. Attributes can be calibration parameters or state variables, more on this later. Attributes without `in` or `out` specifier are considered local variables. A subsystem has instances of state machines and activities.

Calibration parameters and state variables We outline the difference between state variables and calibration parameters.

- State variables are variables that change overtime.
- Calibration parameters are variables that do not change over time, they are usually set before during the creation of an instance of the formal model. In general, re-calibration requires building a new model.

Calibration parameters are tagged with the metadata `CalibrationParam`, while state variables are attributes that have either the `in` or `out` direction attatched.

Observable variables The PT sends telemetry, which is ingested by the DT framework and used for services such as monitoring, fault detection, root cause analysis, simulation and more. Telemetry is the source of information coming from the PT. In the ExploDTwin framework we foresee state variables of the digital twin formal model being part of the telemetry, or not. In the domain of formal methods we call telemetry variables observable variables, or observables. Variables that are not part of the telemetry are non-observable or unobservable variables. In ExploDTwin, state variables can be tagged by the metadata definition `IsObservable`; these state variables are part of the telemetry.

State machines Each subsystem has zero or more state machines. State machines are of different kind, either platform or mission. Mission state machines are characterized by their transitions being durative.

Durative Transitions In state machines usually time passes only in states and transitions happen instantly, that is, no time passes between states. In the domain of planning it usually appears the concept of activity or action. An activity represents a performance by an agent, that usually has preconditions, post-conditions and effects on the world. Moreover, in the use cases we implemented the concept of activity appears as well. It naturally follows that activities should be treated with special care in our framework.

In the ExploDTwin library we defined the concept of durative transitions, which are transitions labelled by activities, and that have a duration. We decided to introduce durative transition for the following reasons:

- abstract away from the user activities implementation details;
- avoid repetition, since activities all have the same structure except for some parameters;
- have more control over the semantics of our system.

The result is that the DT operator interface is greatly simplified. Follows an example of how the user can declare a durative transition. This example shows a `Drill` subsystem which stays idle, performs the warm up activity after which becomes ready.

```
import ExploDTwinMetamodel::Subsystem;
import ExploDTwinMetamodel::Activity;
import ExploDTwinMetamodel::DurativeTransition;

// A definition of a subsystem
part def DrillSubsystem : Subsystem {

  part warmup_activity : WarmupActivity;

  state drillStateMachine : missionStateBehavior {

    state Idle;
    state Ready;
```

```
    transition : DurativeTransition
      first Idle
      then Ready {
        : act = warmup_activity;
      }
  }
}
// A definition of an activity
part def WarmupActivity : Activity;
```

We strive to make the formal model compatible with the tooling that will eventually be available. This tooling should be able to execute SysML v2 models, using KerML's semantics. With this goal in mind we defined a durative transition as follows.

```
action def DurativeTransition : States::StateTransitionAction {
  part act : Activity;

  accepter.payload : ActivityPerformed;
  bind accepter.receiver = act.activityPerformedPort;
}
```

More specifically, a `DurativeTransition` is a subtype of `StateTransitionAction`. Every transition in a state machine, in SysML v2, is implicitly a usage of `StateTransitionAction`. Intuitively, in the code excerpt above we inject manually what would otherwise do the DT designer.

Semantically, a DurativeTransition is a kind of transition that accepts a message from a port in `act`. In the `Drill` subsystem example above, the system stays idle until it receives a message of type `ActivityPerformed` from `warmup_activity`.

In SysML v2 the semantics of subsystems and activities is that of state machines that communicate through messages. This semantics can also be interpreted as the parallel composition of state machines, while carrying the same meaning. An activity is a state machine, and the event-based semantics enabled by a durative transition becomes the parallel composition between the starting and final state of the durative transition and the activity which labels the durative transition.

Platform and mission state machines We distinguish between platform and mission state machines. A platform state machine is a state machine whose semantics is that of a timed automaton. Transitions are caused only by boolean guards and clocks. The DT designer might use a platform state machine to describe a low level behavior.

Mission state machines are state machines whose transition are driven only by durative transitions (i.e. caused by the completion of activities). Mission level state machines are used by the planner to create an activity plan. Intuitively mission state machines control platform state machines.

1.2 Uninterpreted functions

The user may define functions without knowing its internals. That is, the user knows the parameters needed by such functions, and the desired output, but the body of the function is unknown, and it is evaluated by external tooling.

We provide a metadata definition `UninterpretedFunction` which marks a function that will be evaluated by external tooling.

```
package SubsystemPackage {
  private import ExploDTwinMetadata::UninterpretedFunction;

  // ...

  calc def foo {
    @UninterpretedFunction;
    in in1 : Boolean;
    in in2 : Int;
    return : Real;
  }
}
```

1.3 Data types

1.3.1 Ranges

The library provides a type that represents a bounded range of values. It could be useful in situations where a precise value is unknown, but least and upper bounds are known.

```
abstract attribute def RealRange : ScalarRange {
  attribute left : ScalarValue;
  attribute right : ScalarValue;
  constraint {
    left = right
  }
}
```

1.4 Metadata

We use metadata to tag any kind of statement that might be useful to external tooling. There are some ExploDTwin specific metadata, but the DT designer might decide to add some use case specific metadata. The DT designer might use metadata to bridge the gap there could be between the architectural specification and the formal specification.

An attribute (and other kinds of statements of SysML v2) can be tagged by using a metadata definition.

The list below contains all the metadata annotations offered by the library.

- `UninterpretedFunction`. Functions whose body is unknown and computation is performed by external tooling.

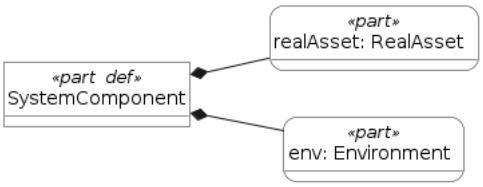

Figure 1: Diagram of the System Component.

- `CalibrationParam`. Attributes that are calibration parameters of a subsystem.
- `ActivityParam`. Used to distinguish between the activity input-parameters and parameters that are owned by the parent subsystem.
- `IsObservable`. Tags subsystem variables that are part of the telemetry, or, in other words, part of the observable state.
- `Unplannable`. This is a planning-specific task. If used to mark an activity, such activity is considered an unplannable and it will not be used by the planner.

2 Formal Model Structure

In this section we describe the structure of the formal model. Section 2.1 contains an intuitive explanation of the structure, without any reference to the formal language. Section 2.2 connects the formal structure to the SysML v2 modelling language.

2.1 Formal structure

In the following we describe each component of ExploDTwin, in hierarchical order.

2.1.1 System Component

The System component represents the PT and its relationship with the environment. Figure 1 shows the components of the System Component. It declares the Environment and the Real Asset components. Moreover, it describes state variables, calibration parameters and the necessary connections at top-level.

2.1.2 Real Asset

The Real asset is the container of all the subsystems of the DT. Figure **??** shows the definition of the Real Asset. Moreover, it describes the relationships and dependencies between all the variables owned by the different subsystems. It may own state variables, calibration parameters.

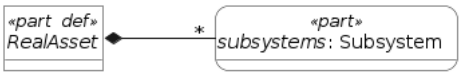

Figure 2: Structure of the Real Asset component.

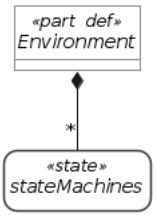

Figure 3: Structure of the Environment component.

2.1.3 Environment

The Environment component describes the environment in which the PT operates, as shown in figure 3. It may contain zero or more state machines, describing different phenomena of the environment. It may declare state variables and calibration parameters.

2.1.4 Subsystem

A subsystem contains a collection of variables, and it may contain a mission, and a platform state-machine. Moreover, each subsystem owns a set of activities. Figure 4 portrays the subsystem and its members, According to its definition in the library.

2.1.5 Activity

Each activity contains a set of variables, some of which are its actual parameters (useful for planning applications). The other variables are inherited by the owning subsystem, and they are necessary to describe conditions and effects of the activity. Figure represents the components of an activity. It is simplified (for example, the state machine that describes the behavior is not portrayed) for the sake of simplicity.

2.1.6 Connections

Whenever different components of the system refer to the same variable, it is necessary to create a connection between the components involved. Three types of connections are possible:

- between two subcomponents variables, and the variables have different port directions;
- between a component and a subcomponent, with different port directions;
- between a component and a subcomponent, with the same port direction.

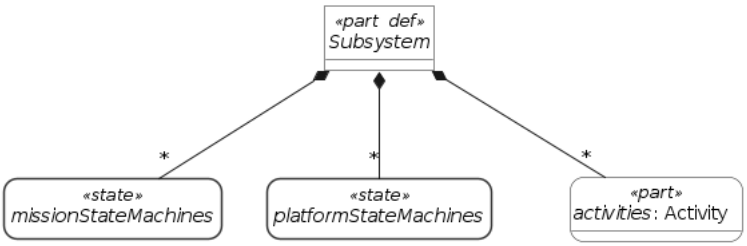

Figure 4: Diagram of the Subsystem component.

Connections between two subcomponents, are instantiated by their immediate parent.

2.2 Formal structure in SysML v2

In this section, we show examples of how a user could define different components of a formal model defined via the ExploDTwin SysML v2 library. We proceed via a top-down approach.

2.2.1 System component and Real asset

The System component and the Real asset only declare variables and connections. The System component connects Environment variables to the Real asset variables. The Real asset connects subsystem components.

2.2.2 Activities

Assuming we have the following uninterpreted functions, and data types defined.

```
enum def DrillStatus {
    enum DrillReady;
}

calc def get_drill_depth {
    @UninterpretedFunction;
    in attribute final_pos: BoundedReal;
    in attribute curr_drill_depth: BoundedReal;
    in lat: Real;
    in lon: Real;

    return: Real;
}

calc def get_drill_pwr_consumption {
    @UninterpretedFunction;
    in attribute final_pos: BoundedReal;
    in attribute curr_drill_depth: BoundedReal;
    in lat: Real;
    in lon: Real;
```

```
    return: Real;
}

part def Drill :> Activity {
    in attribute final_pos: BoundedReal {
        @ActivityParam;
        attribute :>> minValue = 1.342;
        attribute :>> maxValue = 1.952;
    }
    in attribute curr_drill_depth: BoundedReal {
        attribute :>> minValue = 1.342;
        attribute :>> maxValue = 1.952;
    }
    in attribute drill_status: DrillStatus { @IsObservable; }
    in attribute lat: Real;
    in attribute lon: Real;

    calc :>> getLowerBoundDuration { 1 }

    calc :>> getUpperBoundDuration {
        get_upper_bound_drill_duration(
            final_pos,
            lat,
            lon
        )
    }

    constraint :>> precondition {
        drill_status == DrillStatus::DrillReady
    }

    constraint :>> invariant {
        drill_pwr_consumption == get_drill_pwr_consumption(
            final_pos,
            curr_drill_depth,
            lat,
            lon
        )
    }

    constraint :>> postcondition {
        drill_status == DrillStatus::DrillReady
    }

    action :>> posteffect {
        assign curr_drill_depth := get_drill_depth(soil_type, thrust, rot_speed);
    }
  }
```

The user can first define a list of attributes: these attributes may be tagged

with `@ActivityParam`, which means that they are parameters of the activity. The other attributes are necessary to describe conditions and effects.

Pre- and post-effects are lists of assignments.

The user can also provide a lower and upper bound for the duration of the activity. To do so, they must redefine the functions `getLowerBoundDuration` and `getUpperBoundDuration`, these functions are defined in the library as functions with an empty body and empty set of parameters, that return a real value. Thus, the functions if redefined they must return a real value. The example above assumes a call-by-name semantics, i.e. the body of the function is evaluated only when needed. The statement `calc :>> getLowerBoundDuration { 1 }` redefines the computation of the lower bound as the value 1, while for the upper bound duration, it is the result of calling the uninterpreted function `get_upper_bound_drill_duration`.

2.2.3 Subsystems

A subsystem contains variables (state variables and calibration parameters), moreover contains connections with its activities. Of each activity the subsystem connects the end-event (a signal that is generated when an activity finishes), and the components needed to express conditions and effects. A subsystem may have a platform and mission state machines.

```
enum def DRILL_OP_SET {
    enum DRILL_READY;
}

part DrillSubsystem :> Subsystem {

    out attribute drill_status: DRILL_OP_SET;
    out drill_pwr_consumption: real;

    part drill: DrillActivity;

    port drill_done: ActivityPerformedPortIn;

    connect drill_done to drill.activitiPerformedPort
    connect drill_pwr_consumption to drill.drill_pwr_consumption;

    state drill_op_set_SM :> missionsStateMachines {
        // ...
    }
}

part def DrillActivity :> Activity {
    in attribute rot_speed: real { @ActivityParam; }
    in attribute drill_pwr_consumption: real;
    // ...
};
```

2.2.4 Environment

The environment component contains variables and platform state machines.

```
enum def ENV_SAMPLE_OP_SET {
    enum ENV_SAMPLE_EXTRACTED;
    enum ENV_SAMPLE_IN_BOX;
    enum BOX_FULL;
}

part def ExomarsEnv :> Environment {

    out attribute env_sample_status: ENV_SAMPLE_OP_SET;
    out attribute box_load: Integer { @IsObservable; }
    in attribute max_box_load: Integer { @CalibrationParam; }
    in attribute thrust: BoundedReal {
        @IsObservable;

        attribute :>> minValue = -550.0;
        attribute :>> maxValue = 550.0;
    }
    out attribute weight_of_sample: Real;
    in attribute soil_type: soilType;
    in attribute rot_speed: BoundedReal { @IsObservable; }

    port collect_sample_done: ActivityPerformedPortIn;

    state loadSamples :> stateMachines {
        //...
    }
}
```

2.2.5 Mission state machines

A mission state machine is a kind of state machine whose transitions are triggered by durative transitions. In the context of the example presented in section 2.2.3, we show below an example of a mission state machine.

```
state drill_op_set_SM :> missionsStateMachines {

    entry drill_ready;

    transition : DurativeTransition
    first drill_ready
    then drill_ready {
        :>> act = drill;
    }
}
```

The example above defines a mission state machine with only one state. The transition, which is a loop, is triggered by the activity `DrillActivity`. Note that the first statement within the state machine sets the initial state.

2.2.6 Platform state machines

A platform state machine is a kind of state machine whose transitions may be triggered by signals (more specifically, end signals of the activities), and boolean guards. Below an example, of a state machine that monitors the state of the sample, that we assume has already been extracted by the PT using a drill. The sample is then picked up and put in a box only if the weight of the sample does not exceed a parameter. If the weight exceeds `max_box_load` then the state machine goes into an error state.

```
part def ExomarsEnv :> Environment {
    //...

    state loadSamples :> stateMachines {

        entry action {
            assign weight_of_sample := 0;
            assign cstm_load := 0;
        } then env_sample_extracted;

        state env_sample_extracted {
            entry action {
                assign env_sample_status :=
                    ENV_SAMPLE_OP_SET::ENV_SAMPLE_EXTRACTED;
                assign weight_of_sample := get_weight_of_sample(
                    soil_type,
                    thrust,
                    rot_speed
                );
            }
        }

        transition 'sample_picked_up'
            first env_sample_extracted
            accept ActivityEndEvent via collect_sample_done
            then env_sample_in_box;

        transition
            first env_sample_extracted
            if (cstm_load + weight_of_sample > max_cstm_load)
            then box_full;

        state env_sample_in_box {
            entry action {
                assign env_sample_status := ENV_SAMPLE_OP_SET::ENV_SAMPLE_IN_BOX;
            }
        }

        state box_full {
            entry action {
                assign env_sample_status := ENV_SAMPLE_OP_SET::BOX_FULL;
```

```
            }
        }
    }
}

calc def get_weight_of_sample {
    @UninterpretedFunction;

    in attribute soil_type: soilType;
    in attribute thrust: BoundedReal;
    in attribute rot_speed: BoundedReal;

    return: Real;
}
```

The example portrays two transitions, a signal-triggered transition, and a transition triggered by a boolean guard. Note that `ActivityEndEvent` is the kind of the signal that flows into `ActivityPerformedPortIn`.

2.2.7 Connections in SysML v2

Three types of connections are possible:

- between two subcomponents variables, and the variables have different port directions;
- between a component and a subcomponent, with different port directions;
- between a component and a subcomponent, with the same port direction.

In SysML v2 we use two kinds of statements, the `connect` and `bind` statements. We use `connect` to connect two variables with different port directions, `bind` otherwise. According to the list above, three kinds of connections and four kinds of bind statements are possible (connect subcomponent output port to subcomponent input port, connect component output port to subcomponent input port, ...).

```
connect <port> to <port>;
bind <port> = <port>;
```

In the example above we show the syntax of the two statements. The `connect` statement has a direction: it is always an output port that is connected to an input port.

2.2.8 Setters

The ExploDTwin library also provides *setters*. When a component of the formal model performs side-effects on a variable owned by a different component, it does not do it directly on the variable, but by sending a request via a setter port. Intuitively, a setter allows a connection between two components, where one exposes the setter and the other consumes it. A setter is paired to a variable

and a component exposes a setter only if external components are allowed to perform side-effects on said variable.

Note that in ExploDTwin side-effects are performed only by the effect part of the activities. Follows an example where two components are provided, a `Robot` subsystem and a `Move` activity. The robot performs `Move` activities to go from one location to another. The activity performs side-effects on the `current_location` state variable.

In particular, the an action `setter_current_location` is defined, and its meaning is that whenever anything passes through the `set_current_location` port, the `current_location` value changes accordingly. The message is received whenever the activity performs its post-effect.

```
private import ExploDTwinMetamodel::Subsystem;
private import ExploDTwinMetamodel::Activity;
private import ExploDTwinMetamodel::Setter;
private import ExploDTwinMetadata::ActivityParam;
private import ExploDTwinMetadata::IsObservable;
private import ScalarValues::Integer;

enum def LOCATIONS{
    // ...
}

port def SetCurrentLocation {
    in current_location: LOCATIONS;
}

port def SetBatteryLevel {
    in battery_level: Integer;
}

part def Robot :> Subsystem {

    out attribute current_location: LOCATIONS;


    part act_move: Move :> activities;

    port set_current_location: SetCurrentLocation;
    action setter_current_location : Setter {
        action trigger { accept value : LOCATIONS via set_current_location; }
        then assign current_location := trigger.value;
    }

    state example_state_machine :> missionStateMachines {
        //...

    }

    connect current_location to act_move.current_location;
```

```
    connect act_move.set_current_location to set_current_location;
    connect act_move.set_all_included_activity_id
        to set_all_included_activity_id_act_move;

}

part def Move :> Activity {
    in attribute location_to: LOCATIONS { @ActivityParam; }

    port set_current_location: ~SetCurrentLocation;

    action :>> posteffect {
        send location_to via set_current_location;
    }
}
```

2.2.9 Uninterpreted functions

An uninterpreted function is a function that is computed by an external component, such as a machine learning model. The user defines the name, the parameters and the return type. A *calc* definition is an uninterpreted function only if it is tagged by `UninterpretedFunction`.

```
calc def get_upper_bound_drill_duration {
    @UninterpretedFunction;
    in attribute final_pos: BoundedReal;
    in attribute curr_drill_depth: BoundedReal;
    in lat: Real;
    in lon: Real;

    return : Real;
}
```

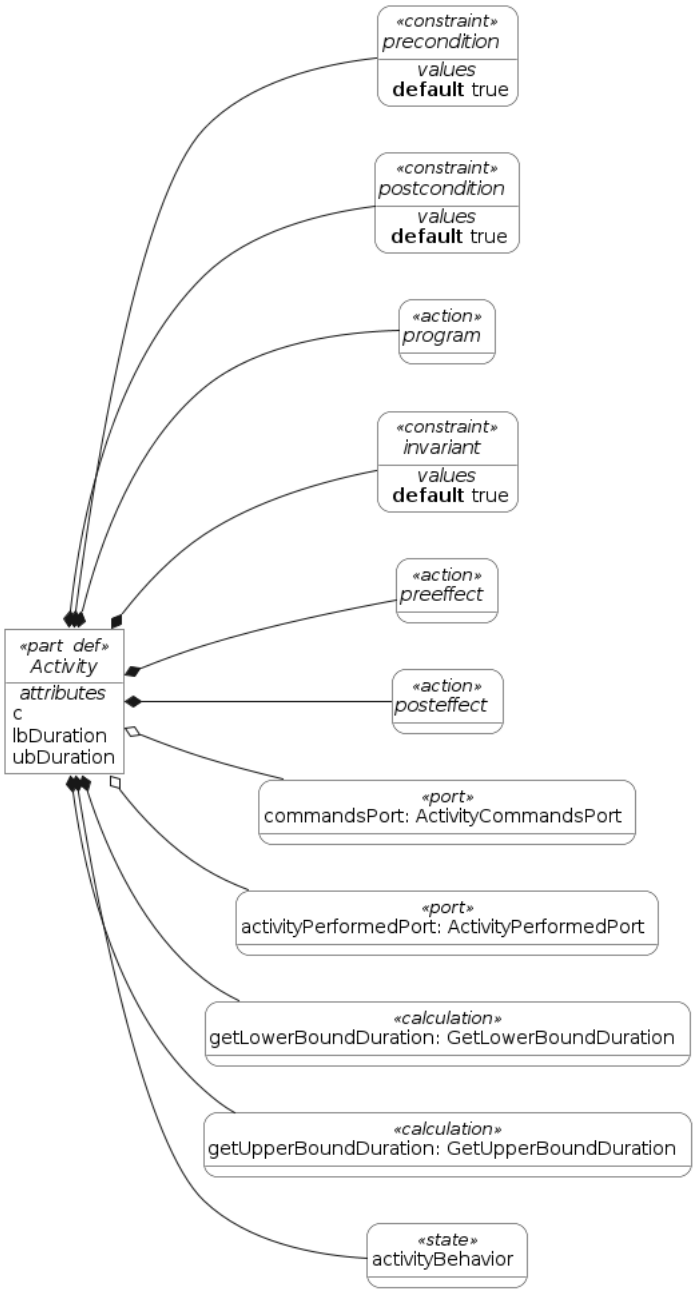

Figure 5: Diagram of the activity.