camunda / **camunda-bpm-dev-docs** Internal

Code Pull requests Actions Security Insights Settings

master

camunda-bpm-dev-docs / concepts / **PVM-Execution-Tree.md**

srazavi correct image insertions **History**

2 contributors

Raw Blame

161 lines (91 sloc) | 12.2 KB

In the following sections, we will dig a little deeper into the engine's internal representation of process instances, the execution tree and its properties.

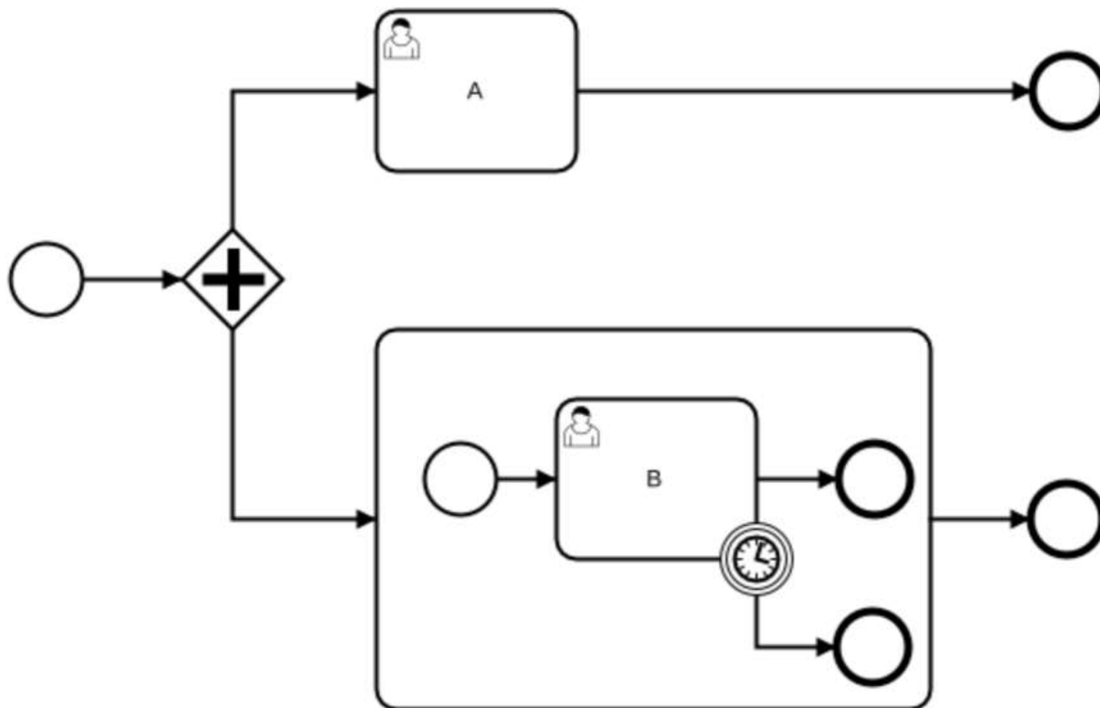
Introduction

In order to execute processes, the process engine has a hierarchical representation of process instances. This representation consists of so-called *executions*. Whenever process execution reaches a wait state, the hierarchy of executions (in the following also referred to as *execution tree*) is persisted to the database table `ACT_RU_EXECUTION` and read from there when execution continues. The tree roughly corresponds to the hierarchy of active BPMN activity instances but is not the same. The following describes how an execution tree is structured and what an execution represents.

On Process Definitions and Activities

Before considering runtime state, let's define the notion of *activities* the process engine has. These include BPMN activities, gateways, events, as well as the process definition itself. Activities may be scopes or not. An activity is a scope if it is a variable scope according to BPMN and Camunda (examples: process definition, subprocesses, multi-instance activities, activities with input/output mappings) or if it defines a context in which events can be received (examples: any activity with a boundary event, any activity containing an event subprocess, any catching intermediate event, any event-based gateway).

Consider the following example:



Out of these, the following activities are scope activities:

- The process definition (variable scope)
- The subprocess (variable scope)
- Activity **B** (defines a receivable event)

The following activities are not scope activities:

- All start and end events
- The parallel gateway
- The boundary event
- Activity **A**

The execution tree's structure and properties depend on whether the current activities are scopes or not.

Executions and Their Properties

An execution is an entity responsible for executing a part of a process instance. Many other instance-related entities are linked to it. These are:

- Timer jobs that reference the execution they trigger
- Event subscriptions (e.g., message) that reference the execution they trigger
- Variables that reference the execution they belong to

Representing a process instance by more than one execution (in fact a tree structure of executions) allows to manage the mentioned related entities conveniently. For example, when an execution is removed, e.g., because it reaches an end event, all related variables that reference it can be easily removed while variables referencing other, still active executions are kept. In short, the lifetime and validity of an execution defines the lifetime and validity of all the entities that reference it.

In the hierarchical structure of executions, different executions have different roles. These are reflected by a set of properties. The basic execution properties are the following:

- *isScope*: A scope execution is responsible for executing a scope activity.
- *isConcurrent*: A concurrent execution is responsible for executing an activity concurrently to another activity in the same scope activity (e.g., two parallel tasks in the same subprocess)
- *isActive*: An execution is active if it is currently executing a plain activity (an activity that does not contain activities itself) or transitioning from one activity to the next.
- *isEventScope*: An event-scope execution is saved for executing compensation. Such an execution must be kept, because BPMN ensures that compensation is executed with a snapshot of the variables at the time of the compensation creation. In the following, we consider only executions that are not event scope executions.
- *activityId*: The id of the activity currently executed by the execution. *Executing* means that the execution is entering the activity, leaving the activity, or performing the activity's actual work. It does not mean waiting for completion of child executions.
- *activityInstanceId*: The id of the activity instance currently executed by this execution.

Execution Tree Patterns

Regardless of how complex the structure of a process instance is, the execution tree is composed of four basic patterns. By combining these patterns, an execution tree can represent arbitrary states of BPMN processes. The patterns are the following:

- Single No-scope activity
- Concurrent No-scope Activities
- Single Scope Activity
- Concurrent Scope Activities

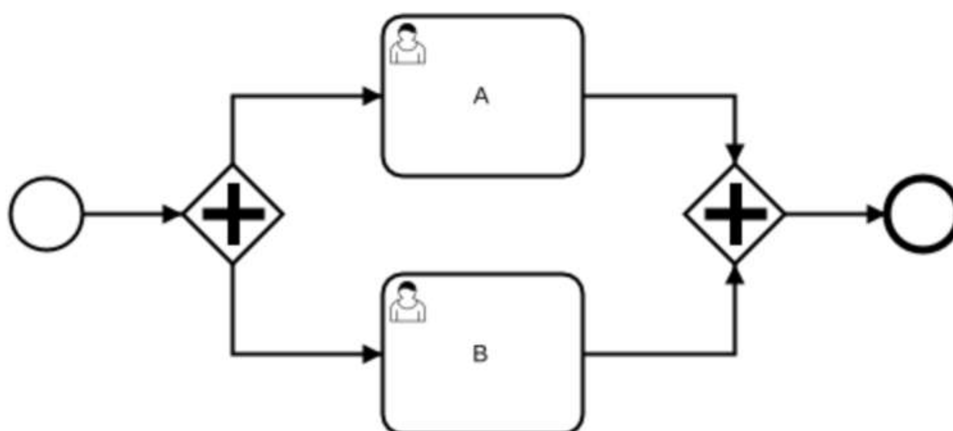
Single No-scope Activity

Activity **A** is not a scope, because it is not a variable scope according to BPMN and has no boundary event.

When **A** is active, the process instance is represented by the following execution tree:

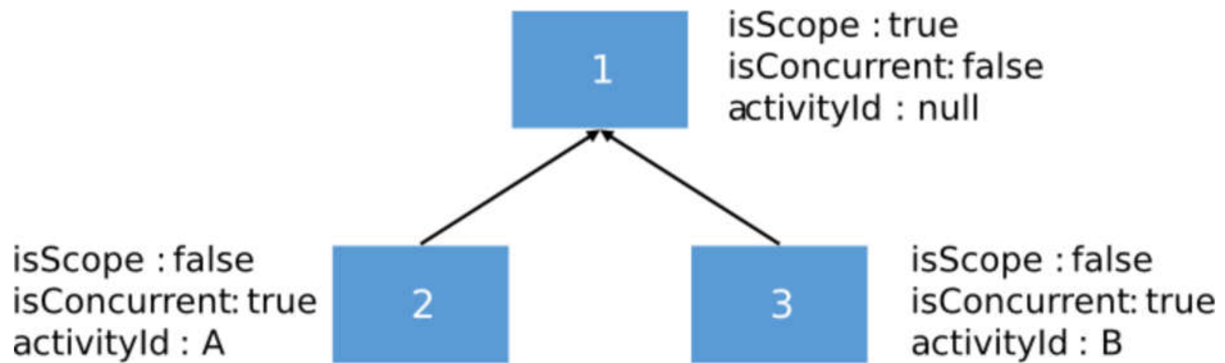
There is a single execution **1**. **1** is the process instance itself. That means, it is responsible for executing the process definition. The process-definition is a scope, so the attribute *isScope* is *true*. **1** is currently executing the activity **A**, which explains the *activityId* setting.

Concurrent No-scope Activities



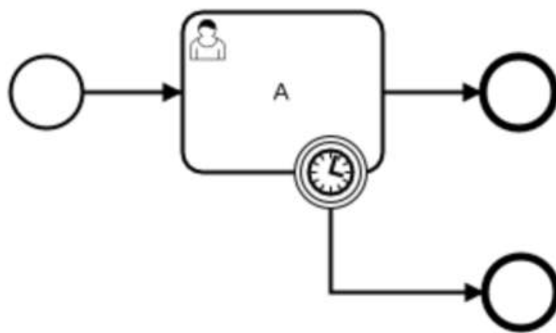
Again, activities **A** and **B** are not scope activities.

When both **A** and **B** are active, the execution tree has the following structure:



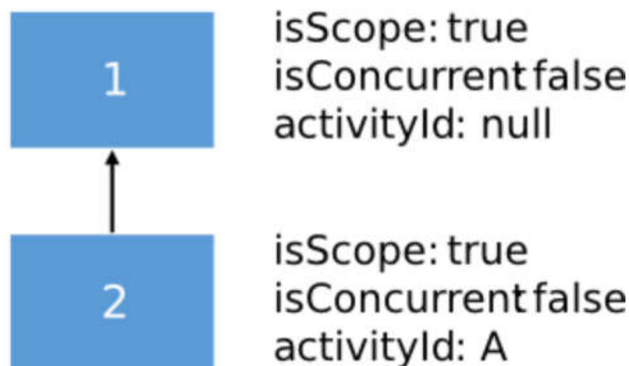
Now, there are three executions. Starting from the root, execution **1** is again the process instance and a scope because the process definition defines a variable scope. It has two children, **2** and **3**. Each of these is concurrent (*isConcurrent* is *true*) and the activities they execute are not scopes (*isScope* is false). Execution **1** has no *activityId* because it is not responsible for actively executing an activity. Instead, **2** and **3** have a non-null *activityId* instead.

Single Scope Activity



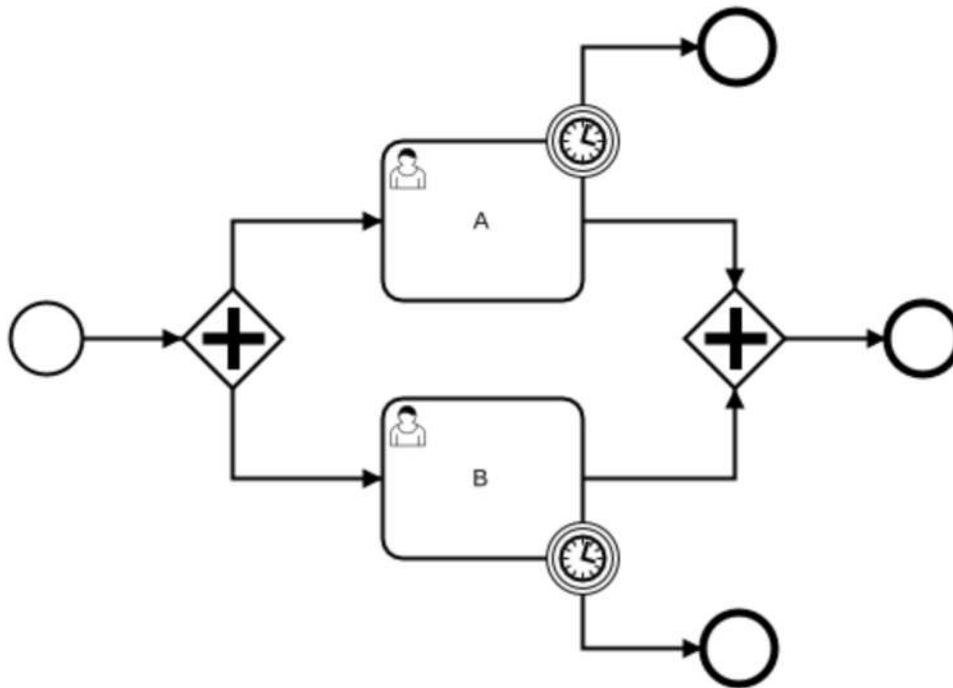
This time, activity **A** is a scope activity. Due to the boundary event, it defines a context in which events can be received. This context is valid for the lifetime of the activity instance. It therefore requires an extra execution.

Thus, when **A** is active, the execution tree looks as follows:



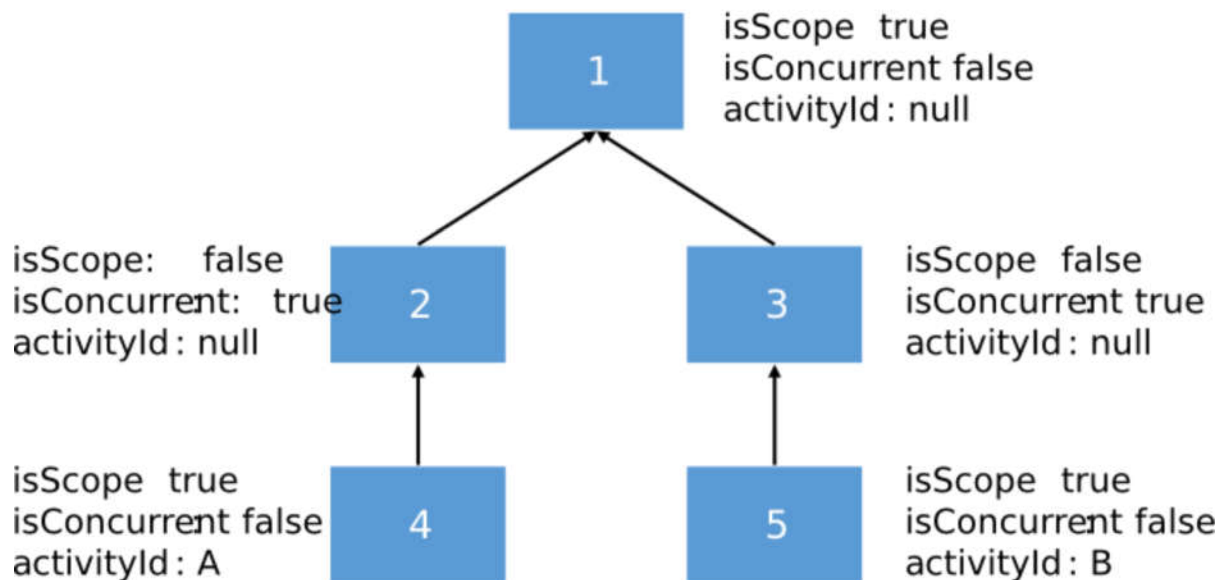
There are two scope executions. Execution **1** is the scope execution for the process definition scope while **2** is the scope execution for activity **A**. Having two scopes is important, because when activity **A** completes, only the event subscriptions of execution **2** should be removed.

Concurrent Scope Activities



Both, **A** and **B**, are scope activities and located in the same parent scope activity.

When both are active, the execution tree is the following:



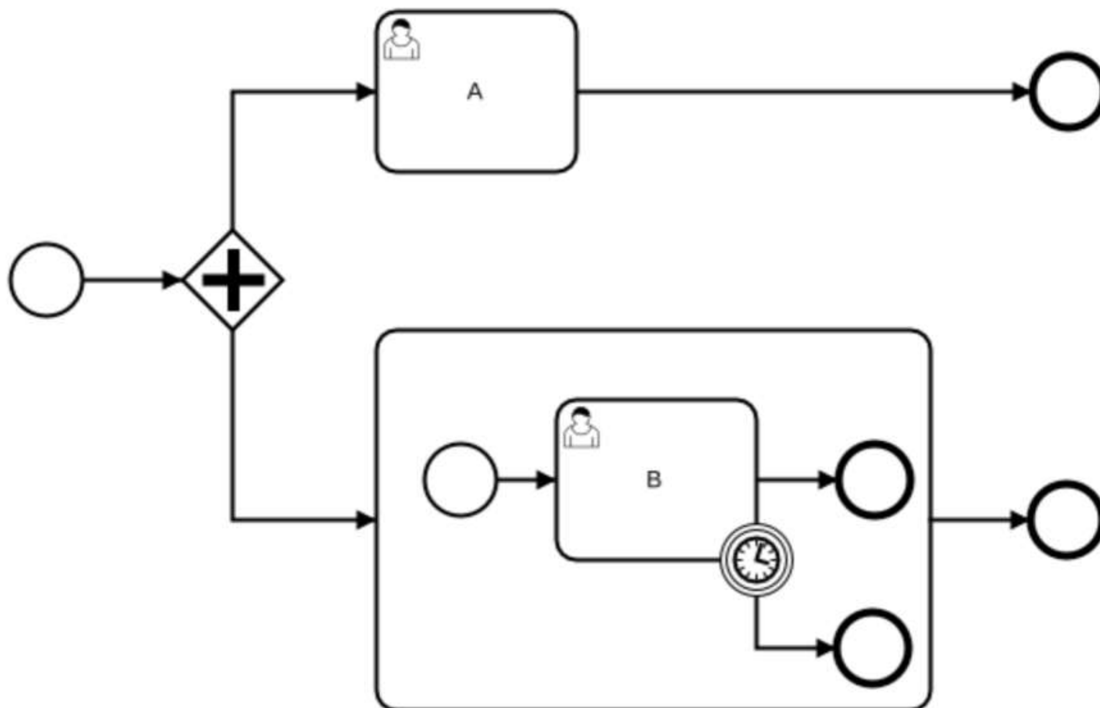
The executions **2** and **3** are just there to represent the concurrency in the scope of **1**. For execution of the activities, the scope executions **4** and **5** are responsible.

Generality of Patterns

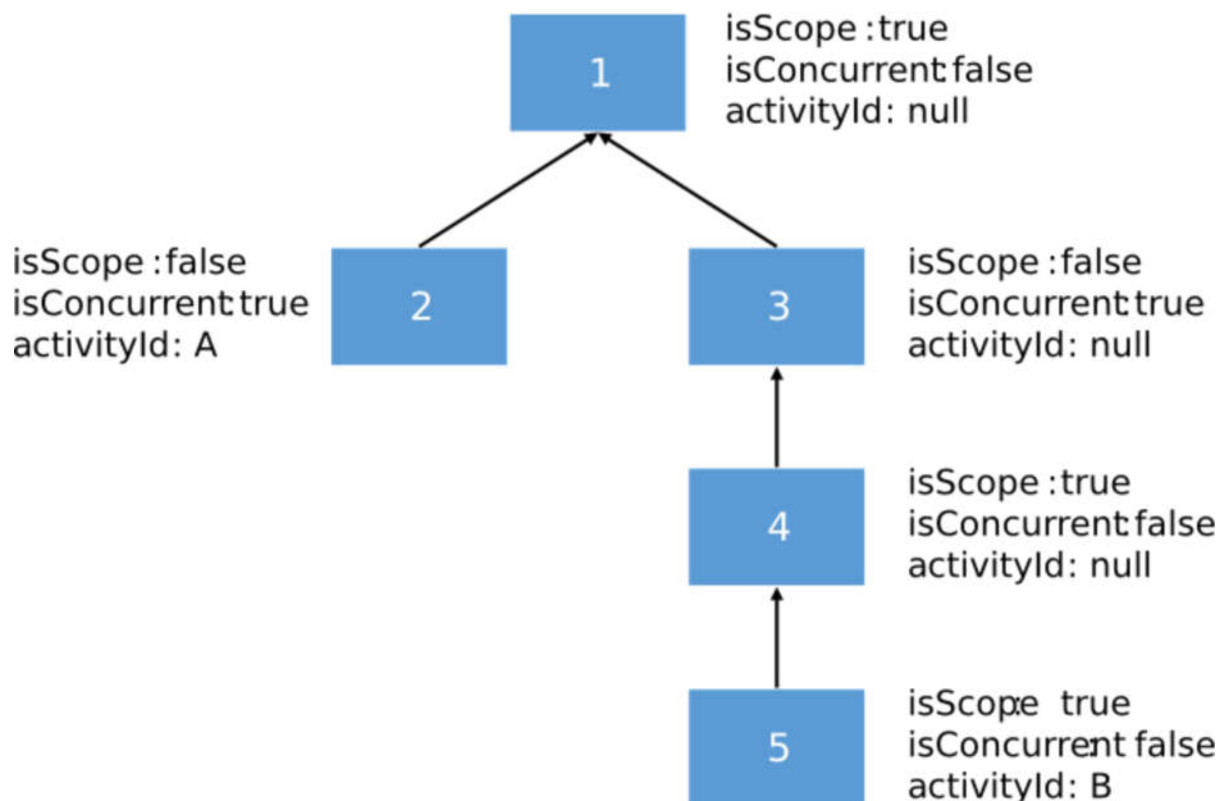
As mentioned above, there are different kinds of scope and none-scope activities. The above patterns are not specific for plain user tasks and user tasks with timer boundary events. Instead, they apply to all cases of non-scope activities (represented by plain user tasks) and scope activities (represented by user tasks with boundary events). In order to apply these patterns to different types of activities, you have to know whether they are scopes or not. Then you can apply the according pattern. For example, for a single task with an input/output mapping (i.e., a scope task), the execution tree looks exactly the same as in the pattern Single Scope Activity.

Composition of Patterns

With arbitrary structures of activities and nesting in subprocesses, the execution tree becomes more complex than the four patterns shown above. However, the tree is only a composition of these patterns. Let's look at an example:



When both activities, **A** and **B**, are active, the tree looks as follows:



Notice how the execution structure of **1**, **2**, **3**, and **4** is a mixed instance of the patterns Concurrent No-Scope Activities and Concurrent-Scope Activities. Execution **4** is responsible for executing the subprocess scope. Since activity **B** is a scope, **4** has a child execution **5**. This is an instance of the pattern Single Scope Activity.

In general, each scope execution in a complex tree is the root of a pattern instance. Here, these scope executions are **1** and **4** with the above-mentioned patterns.

Execution Property Invariants

There are certain invariants for a consistent execution tree. The following statements should always hold:

- The process instance is always the root execution
- The process instance is always a scope execution
- *isScope* and *isConcurrent* are mutually exclusive
- In an execution tree excluding event-scope executions (*isEventScope*), all leaves have a not- `null` *activityId*. All other executions have a `null` *activityId*

If you understand why these invariants hold, you have very likely understood the contents of this chapter :)

The role of the Process Virtual Machine (PVM)

Above, we have considered the static execution tree at a specific point in time. Of course, this tree changes during the course of a process instance, for example when activities start or complete, or a parallel gateway is executed. This is the task of the Process Virtual Machine (PVM). It is responsible for transforming the execution tree such that it always represents the current execution state according to the four patterns. For example, before starting the execution of a scope activity, the PVM makes sure to create a new child execution of the current execution and sets the properties accordingly.

Activity Instances and Executions

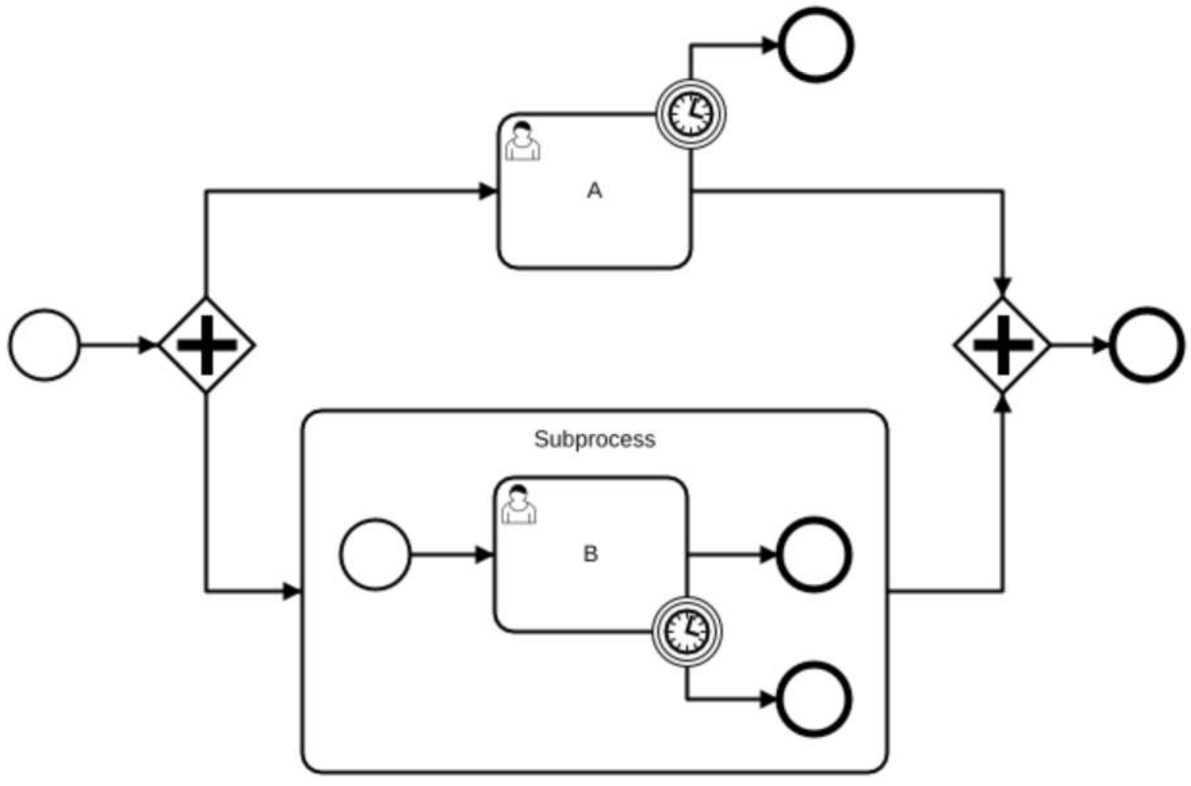
Since the PVM execute executions instead of activity instances, each execution must provide a reference of the activity instance that is currently executed by this execution. The reference is set as property *activityInstanceId* at the execution and is used to create the activity instance tree of the execution.

When an execution starts an activity, the activity instance id of the activity is generated and is set as property of the execution. When the execution leaves the activity, the property of the activity instance id is cleared.

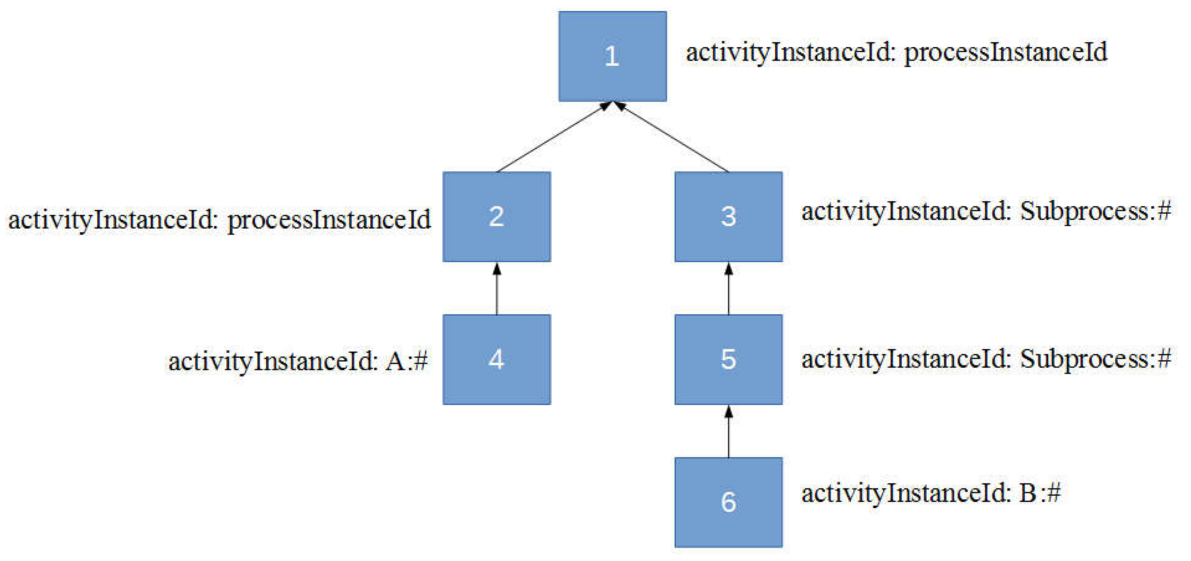
The activity instance id is set on an execution following the first matching rule:

- If the execution has no child executions, the execution has the activity instance id of the activity that is currently executed by this execution.
- If the execution has child executions and execute a composite activity (e.g. subprocess, multi-instance activity), the execution has the activity instance id of the composite activity and the activity instance id is also set to the parent execution, if exists.
- If the execution is the process instance, the activity instance id is the id of the process instance.
- If the execution does not match the above patterns, the execution takes the activity instance id of the parent execution.

For Example:



When both activities, **A** and **B**, are active, the execution tree looks as follows:



Both executions **4** and **6** have no child executions and have the activity instance id of the user task **A** and **B**. The executions **3** and **5** have the same activity instance id because **5** execute the composite activity **Subprocess**. **1** is the execution of the process instance and has the id of the process instance as activity instance id. The execution **2** does not match the other patterns and take the activity instance id of the parent execution **1**.