**LASHM BPM Engine Implementation Concepts**

***Revision #2 10/8/16***

**Primary BPM Components**

* **EPA Base** This actually represents *more* than implied by its title. It encompasses much of the overall BPM Engine workflow including
  + **Input (**AI/CommonSchema**) events handling**
    - Hub Input Events Reader
    - Spark Streaming logic for reading
    - Json validation, error checking and reporting
    - Conversion of JSon to CommonSchema scala structures
  + **Event Partitioning by Correlation Id**
    - Partitioning of incoming events by Correlation Id
    - Distribution of events to specific BPM Process Instance according to Correlation Id
    - Activation and/or creation of BPM process instances by Correlation Id
  + **Persistence and Fault Tolerance**
    - Fault tolerance and persistence of state
    - Dehydration and re-hydration of evicted BPM processes
      * Store and retrieve BPMStates
  + **Output (Derived) Events Handling**
    - Accept emitted DerivedEvents (DE’s) from the Bpm Engine
    - Selectively write DE’s to specific output EventHub’s according to configured rules
      * Presently based on the DerivedEventType of the DE

So all in all the EPA Base is really “**anything (**and everything) **between the incoming and outbound EventHubs except the bpm graph processing / state transition handling and the associated generation of DerivedEvents**.

* **BPM Engine** A single JVM process that combines a provided Camunda BPMN process graph with set of rules to drive the following types of behaviors:
  + State transitions among the nodes of the Graph
  + Generation of Derived Events
  + Logging of exceptions and Errors
  + Timeout management and associated Derived Events generation
  + Tracking and Logging of process state and transitions to support precise queries into processing behavior and history

**BPM Engine Inputs**

\*

The **BPM Engine Static Inputs** include two types of items:

**BPM Graph** : the BPM Nodes and links among them obtained by parsing the Camunda \*.bpmn file and translating to internal BpmNode elements

* **Configuration**: Any variables that are independent of the BPM Graph or Processing State yet which impact those two items
  + These include
    - Process Level Variables. E.g. CorrelationId
    - Environment Variables E.g. Hostname, File System locations

The **BPM Engine Dynamic Inputs** are relevant to BPM Processing at a given moment in time and optionally applied to a single incoming Event . These include two types of items:

* **BPM State** A collection of all conditions both past and present describing conditions specific to a given BPM Instance. These conditions support three types of operations:
  + Queries about current status of the BPM Instance.
    - E.g. Which Nodes of the graph are currently active?
    - Which Nodes have already been processed?
    - Which event(s) are we presently waiting upon?
    - Which events have already processed?
    - Which DerivedEvents have been emitted?
    - Which exceptions have been logged (TODO: should add this..)
  + **Appendix A** showsthe current collection of fields: this list *will* grow
* **AI Event** The incoming AI Event in **CommonSchemaEvent2** format

**BPM Graph Structure**

**BPM Graph Node Types** Graph Nodes are divided into two categories:

* **Event Nodes**: Nodes that expect an external AI Event in order to Advance
* **Internal Nodes**: Nodes that provide graph conditional or branching logic but do not directly consume Events. These are further divided into:
  + Start/End Nodes: these nodes are paired with another node in the Graph:
    - **StartNode**: paired with another node in the Graph that represents the single ***Destination Node*** of the StartNode
      * Presently the other node must be an EventNode. TODO: support InternalNode as well
    - **EndNode**: paired with the last node in the graph representing the single ***Source Node*** of the EndNode
      * Presently the other node must be an EventNode. TODO: support InternalNode as well
* **Branching Nodes** (also known as / **Gateway Nodes**) Provide branching logic – i.e. Fork/Join. Represented by:
  + **Parallel Gateway Node** (aka **AND** Node)
    - **Splitter:**  allows parallel paths of execution within the Graph
    - **Joiner:** Requires ***all*** of its Source Nodes to satisify their “canAdvance()” method before allowing advancement
  + **Exclusive/XOR Gateway Node** (aka **OR** Node)
    - **Splitter:**  allows parallel paths of execution within the Graph
    - **Joiner:** Requires ***any***  of its Source Nodes to satisify their “canAdvance()” method before allowing advancement
* **Conditional Nodes:**  Allow advancement *to a single destination node* based on *Node Rules*
  + (Near..) Future enhancement
* We presently support the following types

**BPM Processing**

Bpm Processing consists of handling Tick invocations – with or without an associated Input/AI Event - and emitting a (possibly-empty) Set of output Derived Events. The considerations in this processing are outlined in this section.

A **BPM Tick** refers to a single “tick” of an abstract system clock that results in the BPM Engine running its internal ***BPM Graph Processing*** logic. If an AI Event were present then the full graph processing logic is applied.

The **No-Event BPM Tick** is a BPM Graph Processing invocation without any associated incoming *AI Event***.** In this case only temporal processing – i.e. Timeout calculations – are run.

The **BPM Graph Processing** refers to the logic applied during each ***BPM******Tick****.* These include:

* Reacting to the input event (if provided) by each of the *Active Nodes*
* Applying timeout calculations to Active Nodes
* Calculating Node Transitions
* Generating output Derived Events
* Calculating the updated State
* Returning the Updated State and any generated Derived Events

The specific rules for Tick() processing are encoded as *Intransitives* that may be seen In **Appendix B**

**Calculating Node Transitions**

First let us describe the *general* rules for Node transitions based on the Node type.

**BPM Node Transition Rules**: Given the

* + (Configuration, BPM State, AI Event) as inputs in the case of BPM Tick
  + or (Configuration,BPM State) for a No-Event Tick
  + Determine if the internal conditions required to satisfy “canAdvance” on any of the Active Nodes were now in place. If so then perform Node Transition(s) to one or more of the **Destination** nodes as follows:
  + **Event Nodes** (“Catch Nodes” in Bpmn parlance): advance *undconditionally*.

We will eventually need to support conditional advancement via **Conditional Event(**/Catch**) Nodes.**  The conditions would be Boolean combinations of value checks against the known (Configuration.BpmState, Event) inputs

* + **Parallel Gateway Nodes** (aka “AND” gate”) **:** advance only if ***all*** *destination* nodes signal “canAdvance” (/satisfied) as true.
  + **Exclusive/XOR Gateway Nodes** (aka “OR” gate”) **:** advance if ***any***of the *destination* nodes signal “canAdvance” (/satisfied) as true.

**BPM Node Transition Calculations:**

Node transitions are affected by following Topological characteristics:

* ***Branching***:
  + multiple Destination Id’s for a given Node on the Split
  + multiple Source ID’s on a given Node for a Join
    - Must distinguish between an AND (parallel gateway) / OR (Exclusive Gateway) Join
* ***Cascaded Branching***
  + A Destination Node of a Branching Node is another Branching Node
* ***Feedback Loop* / *Graph Cycle***
  + A Node contains a Source Node having the original Node as one of its Ancestors

How do we handle these cases? The basic approach is to use the Destination Nodes and/or Source Nodes and recursively traverse them to determine whether the given Node meets a given condition. In addition we have a “feedback Nodes” set on each Node that is a precalculated value of its feedback based Source Nodes.

**BPM Outputs: Derived Events** (“DE” ‘s)**:**  These are the primary outputs of the BPM Engine. They are represented by scala case classes of ***CommonSchemaEvent2*** that may be directly transformed by Jackson to JSON events in the expected format.

* Following are the emitted DE’s
  + <ProcessLabel>.**Created** Emitted when the StartEvent in the process were received.
  + <ProcessLabel>.**InProgress** Emitted under controversial conditions – i.e. subject to change.

Presently: InProgress is emitted

* + - when an EventNode is first activated.
    - If an EventNode were re-activated after a *TimeoutWarning*
  + <ProcessLabel>.**Completed** Emitted when the End Event in the process were received. Note: the EndEvent itself is *not* an EventNode: instead the last EventNode in the graph points to it as its single Destination Node.
  + <ProcessLabel>.**TimeoutWarning** Emitted if an EventNode were delayed in receiving its anticipated message – and according to the following calculation

CurrentEvent.id == ActiveNode.Id

AND (CurrentEvent.BizTime – PriorEvent.BizTime) >= PriorEventNode.ExpectedDuration

* + <ProcessLabel>.**SystemTimeoutWarning** Emitted if an EventNode were delayed in receiving its anticipated message – and according to the following calculation

(CurrentTick.SystemTime – PriorEvent.SystemTime) >= PriorEventNode.ExpectedDuration

* + <ProcessLabel>.**ProcessTimeoutError** Emitted if the *overall process runtime* exceeds a Camunda Model specified maximum value – and according to the following calculation

CurrentEvent.BizTime – ***First***Event.BizTime) >= Configuration.ProcessVariables.MaximumProcessTime

* + <ProcessLabel>.**SystemProcessTimeoutError** Emitted if the *overall process runtime* exceeds a Camunda Model specified maximum value – and according to the following calculation

(CurrentTick.SystemTime – **First**Event.SystemTime) >= Configuration.ProcessVariables.MaximumProcessTime

* **Error/Exception Logging** Following conditions will result in logged messages:
  + **Unrecognized Event Id**: the event Id is not included in the Graph nodes Id’s
  + **Ineffectual Event**: an Event were received either:
    - **Out-of-Order** The event cameToo soon. I.e it is destined for a Node that is not yet in ActiveNodes
    - **Redundant**: i.e.it is destined for a Node in the Already Processed set (and thus *not* in the Active set)
    - **Zombie** The overall Process Instance has already reached a ***Terminal State*** (either Completed or Error)
  + **Data Errors**: e.g. an unparseable timestamp
  + **Rules Driven Exceptions**: when we have Rules Processing they may elect to emit Exceptions under prescribed conditions (*Future Enhancement)*

**Appendix A: BpmState Fields** (as of 10/8/16)

**case class** BpmState(  
 correlationId: CorrelationId[\_],  
 status: StatusInfo = *StatusInfo*(New, 0),  
 isStatusUpdated: Boolean = **true**,  
 updatedAt: Long = System.*currentTimeMillis*,  
 createdAt: Long = System.*currentTimeMillis*,  
 expectedCompletionAt: Option[Long] = None,  
 closed: Option[Long] = None,  
 closedReason: Option[BpmStatus] = None,  
 statusHist: Vector[StatusInfo] = *Vector*.*empty*[StatusInfo],  
 events: Seq[BpmEvent] = *List*.*empty*[BpmEvent],  
 activeNodes: Map[String, BpmNodeState] = *Map*.*empty*[String, BpmNodeState],  
 processedNodes: Map[String, BpmNodeState] = *Map*.*empty*[String, BpmNodeState],  
 nodeTransitions: Seq[NodeTransition] = *Seq*.empty[NodeTransition],  
 props: mutable.Map[String, Any] = mutable.Map.*empty*[String, Any],  
 archivedNodes: Seq[(Int, Set[BpmId])] = *Seq*.empty[(Int, Set[BpmId])],  
 **var** \_lastEventInfo: Option[EventInfo] = None,  
 **var** nextEventIds: Set[BpmId] = *Set*.*empty*[BpmId],  
 **val** processStartTime: Long = System.*currentTimeMillis*,  
 **var** inProgressMap: mutable.HashMap[BpmId, Boolean] = **new** mutable.HashMap[BpmId, Boolean],  
 **var** firstEventInfo: EventInfo = *EventInfo*(*StartEvent*, Long.*MaxValue*, System.*currentTimeMillis*, Long.*MaxValue*),  
 **var** derivedEvents: Seq[Seq[DerivedEvent]] = *Seq*.empty[Seq[DerivedEvent]],  
 **var** exceptions: Seq[Seq[(String, Throwable)]] = *Seq*.empty[Seq[(String, Throwable)]],  
 **var** feedbackCounter: Int = 0) {

**Appendix B: Invariants Checks for the Tick():**

**Inputs:** Optional AI Event.  
 BpmState  
 System clockTime "now"  
**Outputs:** Updated BpmState  
 (Possibly empty) Sequence of Derived Events  
  
**Outgoing Invariants:**  
***CanAdvance calculation:*** EventNodes for which their "listenTo" message has been received  
 Parallel (/AND) GatewayNodes for which ALL of their Source Nodes canAdvance  
 XOR GatewayNodes for which ANY of their Source Nodes canAdvance are canAdvance  
  
***ActiveNodes:*** All EventNodes having !canAdvance but their SourceNode has canAdvance  
 All GatewayNodes with canAdvance but one or more DestNodes are !canAdvance  
 EventNodes for which their own "listenTo" Event has been received are in the ProcessedNodes set - regardless of state of  
 their DestinationNodes  
  
***ProcessedNodes:*** All EventNodes with canAdvance  
 All GatewayNodes for which all of their DestNodes are canAdvance  
  
Define ActiveOrProcessed === InputState.ActiveNodes Union InputState.ProcessedNodes  
  
***FirstEventInfo:*** Fields are equal to those of incoming Event when:  
 FirstEvent received and FirstEvent not in ActiveOrProcessed  
  
***LastEventInfo:*** Fields are equal to those of incoming Event when:  
 ANY eventNode were activated by the incoming Event  
 i.e. InputState.activeOrProcessed != OutputState.activeOrProcessed  
  
***DerivedEvents:*** Created:  
 FirstEvent received and FirstEvent not in ActiveOrProcessed  
 InProgress[Event]:  
 Nodes(Event) not in ActiveOrProcessed  
 Completed:  
 LastEvent received and LastEvent in InputState.ActiveNodes  
 TimeoutWarning  
 CurrentEvent.id == ActiveNode.Id  
 AND (CurrentEvent.BizTime – PriorEvent.BizTime) >= PriorEventNode.ExpectedDuration  
 SystemTimeoutWarning  
 (CurrentTick.SystemTime – PriorEvent.SystemTime) >= PriorEventNode.ExpectedDuration  
 ProcessTimeoutError  
 CurrentEvent.BizTime – FirstEvent.BizTime) >= Configuration.ProcessVariables.MaximumProcessTime  
 SystemProcessTimeoutError  
 (CurrentTick.SystemTime – FirstEvent.SystemTime) >= Configuration.ProcessVariables.MaximumProcessTime