BPE: A Lightweight Native BPMN Implementation Preserving Clean Process Semantics

Maxim Sokhatsky
Synrc Research

May 23, 2026

Abstract

BPE (Business Process Engine) is a compact, production-grade workflow engine written in Erlang/OTP that implements a clean subset of BPMN 2.0. Unlike traditional BPMN engines such as Camunda, where developers are forced to attach scripts and listeners to gateways and sequence flows (pre/post events), BPE treats the process as a natural sequence of actions executed within a single semantic step. This approach dramatically reduces complexity while maintaining full persistence, fault-tolerance, and auditability.

Contents

1 Introduction

BPE is a lightweight business process management engine developed as part of the Synrc stack. It is designed for real enterprise use — particularly in banking, document management, CRM, and telecom systems — where reliability, auditability, and simplicity are paramount.

At its core, BPE is a microkernel that executes native Erlang record-based process definitions combined with Event-Condition-Action logic. The entire engine consists of roughly 400 lines of clean functional code, making it one of the most compact and maintainable workflow engines available.

BPE stores all process state and history in KVS (Key-Value Store), providing strong persistence guarantees and seamless integration with the Erlang ecosystem. Every process instance is a supervised Erlang process with full crash recovery and migration capabilities.

1.1 Key Design Philosophy

The central idea of BPE is to preserve **clean process semantics**.

In most BPMN engines (e.g. Camunda, Activiti, jBPM), the execution model forces developers to split business logic across:

- Sequence flow conditions
- Execution listeners (pre/post)
- Task listeners
- Gateways with attached scripts

This results in fragmented logic that is hard to read, debug, and maintain.

BPE takes the opposite approach: **the process flow is executed naturally in one action**. A task (userTask or serviceTask) defines its module and the engine calls the corresponding Erlang functions directly. The transition to the next step is handled atomically within the same semantic context.

This design aligns with Erlang's native process calculus (Pi-calculus) semantics and provides a much more idiomatic and readable way to express business workflows.

2 Process Calculus

2.1 Objects

Processes are defined as Erlang records:

```
deposit_app() -> #process {
    name = 'Create Deposit Account',
    flows = [
        #sequenceFlow{source='Init',       target='Payment'},
        #sequenceFlow{source='Payment',    target='Signatory'},
        #sequenceFlow{source='Payment',    target='Process'},
        #sequenceFlow{source='Signatory', target='Process'},
        #sequenceFlow{source='Signatory', target='Finish'}
    ],
    tasks = [
        #userTask    {name='Init',       module=deposit},
        #userTask    {name='Signatory', module=deposit},
        #serviceTask {name='Payment',    module=deposit},
        #serviceTask {name='Process',    module=deposit},
        #endEvent    {name='Finish'}
    ],
    beginEvent = 'Init',
    endEvent = 'Final'
}.
```

2.2 Morphisms

- `bpe:start(Process, Options)` — starts a new process instance
- `bpe:complete(PidOrId)` — moves the process to the next step
- `bpe:amend(PidOrId, Document)` — attaches business documents
- `bpe:event(Event)` — sends external events
- `bpe:history(Id, N)` — retrieves execution log
- `bpe:load(Id)` — loads a terminated process for continuation

All operations are atomic and persisted through KVS.

3 Task Taxonomy and Execution Model

At the core of the BPE workflow model are tasks, which act as the vertices of the process graph. BPE defines a structured taxonomy for tasks, enabling both automated computations and human-in-the-loop interactions.

3.1 The Base Task Macro

Every task defined in the system inherits a set of common metadata fields through the `?TASK` macro. In the Erlang record representation, these properties provide the fundamental execution schema for the engine:

```
-define(TASK,
    id      = [] :: [] | list() | binary() | atom(),
    name    = [] :: [] | list() | binary(),
    input   = [] :: list(),
    output  = [] :: list(),
    prompt  = [] :: list(),
    roles   = [] :: list(atom()),
    etc     = [] :: list({term(), term()})
).
```

These attributes serve the following purposes:

- `id`: A unique identifier for the task node.
- `name`: A human-readable label used primarily in UI rendering.
- `input` / `output`: Structured parameter schemas defining the data boundary.
- `prompt`: Configuration details describing what inputs are expected from external interfaces.
- `roles`: A list of Erlang atoms representing user roles authorized to trigger actions on this task.
- `etc`: An open key-value list for custom task metadata.

3.2 Task Classifications

BPE inherits a set of task nodes matching the BPMN 2.0 specification. Each is implemented as a native Erlang record:

- **Start and End Nodes**:

  ```
  -record(beginEvent, { ?TASK }).
  -record(endEvent,   { ?TASK }).
  ```

  Reaching the `endEvent` terminal node instructs the engine to immediately stop execution and terminate the process session.

- **Computational and Interaction Nodes**:
  - `#task{}`: A generic execution node.
  - `#serviceTask{}`: An automated backend service node that executes logic without human intervention.
  - `#userTask{}`: A human-in-the-loop stage. Transitioning to or executing a `userTask` requires authorization checks (typically evaluated via `Module:auth/1`).
- **Communication Nodes**:

  ```
  -record(receiveTask, { ?TASK, reader = [] :: #reader{} }).
  -record(sendTask,    { ?TASK, writer = [] :: #writer{} }).
  ```

  A `receiveTask` halts execution, waiting for a matching external reader event. A `sendTask` handles outbound messaging using a designated `writer` profile.

3.3 Task Action Execution Model

When a task transitions, BPE invokes the process module's execution callback:

```
Module:action({request, From, To}, Proc) -> #result{}
```

The outcome is encapsulated within a formal `#result{}` record:

```
-record(result, {
    type     = reply :: reply | noreply | stop,
    reply    = [] :: term(),
    state    = [] :: #process{},
    reason   = normal :: term(),
    continue = [] :: list(#continue{}),
    opt      = [] :: term(),
    executed = [] :: list(#executor{})
}).
```

This record allows callbacks to update the state, return responses to synchronous callers, request execution termination (`stop`), or delegate background tasks.

4 Event Semantics and Message Lifecycle

BPE contains a generalized event handling subsystem managing synchronous communication, asynchronous execution, and system-wide broadcasts.

4.1 Event Records

Events in BPE are formalized in `bpe.hrl` under four main structures:

- **Synchronous Message Event**:

```
-record(messageEvent, {
    id      = [] :: [] | list() | binary(),
    topic   = [] :: [] | list() | binary(),
    sender  = [] :: [] | term(),
    timeout = [] :: [] | #timeout{},
    name    = [] :: [] | term(),
    payload = [] :: [] | term()
}).
```

  Sent via `bpe:messageEvent/2`, this triggers synchronous processing and blocks the caller until the transition completes.

- **Asynchronous Event**:

```
-record(asyncEvent, {
    id      = [] :: [] | list() | binary(),
    sender  = [] :: [] | term(),
    topic   = [] :: [] | list() | binary(),
    name    = [] :: [] | term(),
    payload = [] :: [] | term()
}).
```

  Dispatched via `bpe:asyncEvent/2` as a non-blocking `gen_server:cast`.

- **System-wide Broadcast Event**:

```
-record(broadcastEvent, {
    id      = [] :: [] | list() | binary(),
    sender  = [] :: [] | term(),
    topic   = [] :: [] | list() | binary(),
    type    = immediate :: immediate | delayed,
    name    = [] :: [] | term(),
    payload = [] :: [] | term()
}).
```

- **Boundary Event**:

```
-record(boundaryEvent, {
    id           = [] :: list() | atom(),
    name         = [] :: list() | binary(),
    prompt       = [] :: list(tuple()),
    etc          = [] :: list({term(), term()}),
```

```
    payload      = [] :: [] | binary(),
    timeout      = [] :: [] | #timeout{},
    timeDate     = [] :: [] | binary(),
    timeDuration = [] :: [] | binary(),
    timeCycle    = [] :: [] | binary()
}).
```

Attached to task boundaries, these events trigger timeouts or redirect execution when specific time bounds are reached.

4.2 Callback Redirection

All events target a matching callback handler in the workflow process module:

```
Module:action({event, Name, Payload}, Proc) -> #result{}
```

This architecture separates routing logic from event-handling behaviors, keeping process execution clean.

5 Gateway Routing and Join Synchronization

Gateways dictate how paths diverge and converge inside a process graph. BPE provides deterministic algorithms for branching, dynamic compilation, and join synchronization.

5.1 Gateway Classification

BPE supports four gateway types:

- **Exclusive Gateway (Split)**: Evaluates outgoing flow conditions in order, selecting the first that returns `true`.
- **Parallel Gateway (Split/Merge)**: Fork/join mechanics. Merging waits for all incoming flows before moving forward.
- **Inclusive Gateway (Split/Merge)**: Evaluates all conditions, spawning concurrent threads for each matching flow. Merging acts as a synchronizing join.
- **Complex Gateway**: Custom developer-defined branch-and-merge algorithms.

5.2 Dynamic XML Condition Compilation

When parsing BPMN 2.0 XML files, BPE compiles formal conditions written inside `<bpmn:conditionExpression>` elements. At load-time, BPE parses the text using `erl_scan` and `erl_eval` to generate Erlang tuples:

- **Compare**: {`compare, Pattern, FieldIndex, Value`} checks if the environment documents matching `Pattern` contain `Value` at `FieldIndex`.
- **Service**: {`service, Fun, Module`} delegates to a custom evaluation function: `Module:Fun(Proc)`.
- **Gateway Block**: {`service, gw_block`} checks Mnesia's `gw_block` table to hold transition.
- **Default**: `[]` evaluates to `true`.

BPE also supports sequence flow transition callbacks, allowing logic invocation during step transitions:

```
{callback, Fun, Module, Arg} % Evaluates Module:Fun({callback,
    Src, Dst}, Result, Arg)
```

5.3 Exclusive Gateway Branching Algorithm

For exclusive gateways, BPE sequentially scans outgoing sequence flows. If none evaluate to `true`, it defaults to the flow defined in the `def` field of the gateway record. If no default is defined, BPE halts execution with a runtime exception:

```
get_inserted(#gateway{id = Name, type = exclusive, output = Out
    , def = []}, _, _, Proc) ->
    case first_matched_flow(Out, Proc) of
        [] ->
            add_error(Proc, "All conditions evaluate to false
                in exclusive gateway without default", Name),
            [];
        X -> X
    end;
get_inserted(#gateway{type = exclusive, output = Out, def =
    DefFlow}, _, _, Proc) ->
    case first_matched_flow(Out -- [DefFlow], Proc) of
        [] -> [DefFlow];
        X -> X
    end;
```

5.4 Join Synchronization Algorithm

Parallel and inclusive gateways acting as merges must block the advancing execution thread until all incoming flows have reached the gate. BPE implements this by backtracking through the execution history logs using the history decrement algorithm:

```
get_inserted(#gateway{type = Type, input = In, output = Out},
    Flow, ScedId, _Proc)
    when Type == inclusive; Type == parallel ->
    case check_all_flows(In -- [Flow#sequenceFlow.id], ScedId)
        of
        true  -> Out;
        false -> []
    end;

check_all_flows([], _) -> true;
check_all_flows(_, #step{id = 0}) -> false;
check_all_flows(Needed, ScedId = #step{id = Id}) ->
    case hist(ScedId) of
        #hist{task = #sequenceFlow{id = Fid}} ->
            check_all_flows(Needed -- [Fid], ScedId#step{id =
                Id - 1});
        _ -> false
    end.
```

6 Engine Concurrency, Locking, and Recovery

BPE operates in a highly concurrent environment. To prevent double-execution, race conditions, and to manage persisted processes, the engine utilizes a set of robust protocols.

6.1 Mnesia Locking Coordination Protocol

To prevent parallel requests from starting concurrent `gen_servers` or overlapping execution cycles on the same process, BPE implements Mnesia-based coordination locks:

1. Every API call (such as `next`, `complete`, or `messageEvent`) generates a monotonic lock ID and registers a `#terminateLock{id = Id, pid = ProcId}` record in Mnesia.

2. Upon call completion, BPE invokes `bpe_proc:terminate_check/5`. If the pending lock count exceeds 500, the engine terminates the session with an error to prevent deadlocks.

3. If a process yields a `stop` command but has pending locks in the queue, BPE converts the stop instruction to a normal reply. This keeps the `gen_server` online to drain outstanding requests.

4. When the last lock is cleared and the process stops, it registers in the `terminateLocks` ETS cache.

6.2 ETS Cache Blocking

When spawning process `gen_servers` (via `bpe:start/2`), BPE checks the `terminateLocks` ETS cache. If a matching process ID is found (indicating it is in the process of shutting down), the starter blocks and waits for a `'DOWN'` monitor message before initiating a new instance.

6.3 Offline Event Reception and Queue Drainage

To conserve system resources, inactive processes are unloaded and kept only in persistent KVS storage. BPE guarantees delivery of events to offline processes via a restoration and queue drainage protocol:

- **Automatic Restoration**: Any API interaction triggers `start(load(ProcId), [])`, which restores state from KVS and spawns a worker if offline.
- **Delayed Event Queueing**: Non-immediate events sent to offline processes are appended to the process's database queue in Mnesia under `/bpe/messages/queue/<ProcId>`.
- **Ping Drainage Cycle**: Upon initialization, BPE's `init/1` schedules a periodic {`timer, ping`} message. When received, the process reads the queued events and executes them sequentially via `process_event/3`. Any leftover queue is drained during the `terminate/2` phase.

6.4 Asynchronous Continuation Framework

For long-running tasks, BPE supports asynchronous continuation, allowing developers to offload computation without blocking the main workflow engine. This is accomplished using the `#continue{}` record inside the `#result{}` output structure:

```
-record(continue, {
    type   = spawn :: spawn | cast,
    module = []    :: [] | atom(),
    fn     = []    :: [] | atom(),
    args   = []    :: list()
}).
```

When returned from a task action callback, the engine automatically triggers the continuation logic, e.g., spawning background Erlang processes or invoking async RPC calls, preserving the main engine thread for fast routing.

7 Conclusion

7.1 Comparison with Camunda and Other Engines

- **Clean Semantics**: BPE executes business logic inside task modules rather than scattering it across listeners and gateways.
- **No XML Hell**: Processes are defined in Erlang terms (or optional DSLs).
- **Native Persistence**: Every step and document is automatically saved via KVS/RocksDB.
- **Supervision & Fault Tolerance**: Built on Erlang/OTP principles.
- **Auditability**: Complete immutable history is maintained automatically.
- **Performance**: Extremely low overhead — suitable for high-frequency financial workflows.

While Camunda excels in visual modeling for large analyst teams, BPE is optimized for engineering teams that value clarity, maintainability, and performance over drag-and-drop modeling.

7.2 Use Cases

BPE has been successfully used in:

- Banking core systems (deposit, credit, payment processing)
- Document workflow and approval systems
- CRM interaction flows
- Telecom service activation and billing processes
- Regulatory compliance workflows

Its ability to handle both human tasks and automated service tasks within the same clean model makes it particularly suitable for hybrid human–machine business processes.

7.3 Last words

BPE demonstrates that a BPMN-compatible workflow engine can be implemented in a remarkably clean and efficient way by staying close to the underlying virtual machine semantics. By executing sequence flows naturally within one action and leveraging Erlang's process model, BPE achieves excellent readability, strong fault tolerance, and production readiness with minimal code.

For projects that prioritize engineering clarity and operational reliability over visual designer tools, BPE offers a compelling lightweight alternative to heavyweight BPM suites.

Documentation and source code: `https://synrc.com`