Producer Actor

Zhifeng Zhang

Oct. 15, 2024

Contents

The Producer Actor

Definition

We can not model the producer as a Mealy machine due to the non-deterministic nature, we need state be able to emit events. This leads to the actor model, which is more suitable for the producer.

Unlike Mealy machines, which are inherently deterministic and reactive—responding strictly to external inputs—the Producer Actor requires the ability to **proactively generate events** based on internal state transitions and actions. This necessitates a model that supports **asynchronous communication**, **state encapsulation**, and **event-driven behavior**, all of which are core characteristics of the Actor Model.

Formally, the producer actor is a sextuple state machine $M = \langle Q, \Sigma, \delta, \lambda, q_0, F \rangle$, where:

- Q: The set of states representing distinct operational modes of the producer.
- Σ: The set of possible events that the producer can handle or emit.
- δ: The transition function dictating how the producer moves between states in response to events.
- λ: The action function defining the events the producer emits based on state-event pairs.
- q_0: The initial state where the producer awaits commands.
- F: The set of accepting states indicating successful operation cycles.

By convention, Σ^* denotes the set of all finite sequences (strings) of events generated by Σ. A word $w = \sigma_1\sigma_2 ... \sigma_n \in \Sigma^* \in \Sigma^*$ is an accepting word if, after processing the entire sequence, the system transitions into one of its accepting (final) states F.

This model conform to the actor model but differ from the Mealy machine in the following way:

1. Output word space Γ is removed.
2. The running input word Σ^* is composed of outside input word and input word that is generated from actions, both inputs conform with Σ.

This sextuple framework aligns with the Actor Model by facilitating **state encapsulation** and **event-driven message passing**. Unlike traditional automata where outputs are separate from inputs, here, the action function λ generates new events within Σ, enabling **dynamic and asynchronous interactions** within the system.

State space Q

The **state space** $Q = \{s_0, s_1, s_2, s_3\}$, where

- s_0 = Idle: The system is idle, waiting for a `FETCH_DATA` command to initiate data retrieval.
- s_1 = FetchingData: The system is actively fetching data from CryptoCompare upon receiving `FETCH_DATA`.
- s_2 = Publishing: Once data is ready (`DATA_READY`), the system publishes the aggregated data to Kafka/Redpanda.
- s_3 = Error: Handles any failures during fetching or publishing, responding to `FETCH_FAILURE` or `PUBLISH_FAILURE`.

The initial state $q_0 = s_0$ and the acceptor $F = s_0$:

- **Initial State** (s_0 = Idle): Awaiting commands to fetch or publish data. This is also the **accepting state**, indicating that operations have successfully cycled back to an idle state.

The input space Σ

$\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6\}$, where

- $\sigma_1 = FETCH\_DATA$: signal to initiate the data fetching.
- $\sigma_2 = DATA\_READY$: notification that data has been successfully fetched.
- $\sigma_3 = FETCH\_FAILURE$: notification that data fetched failed.
- $\sigma_4 = PUBLISH\_SUCCESS$: confirmation that data has been successfully published.
- $\sigma_5 = PUBLISH\_FAILURE$: notification of a failure in publishing data.
- $\sigma_6 = RESET$: signal to reset the actor from an error state.

In our platform, the elements of Σ are termed **events**, which can originate from external user inputs or be generated internally by the actor's actions.

The transition function δ

The transition function δ defines state transitions based on incoming events. Each transition is triggered by an event, leading to a state change that may also result in generating new events through the action function λ.

Transition function $\delta : Q \times \Sigma \to Q$ is defined as

```
<div class="table-caption">
  <span class="table-number">Table 1:</span>
  Transition function $\delta$ for producer actor - a symbolic definition:
</div>
```

current state	event	next state
s_0	σ_1	s_1
s_0	All other σ	s_0
s_1	σ_2	s_2
s_1	σ_3	s_3
s_2	σ_4	s_0
s_2	σ_5	s_3
s_3	σ_6	s_0

```
<div class="table-caption">
  <span class="table-number">Table 2:</span>
  Transition function $\delta$ for producer actor - a descriptive definition:
</div>
```

current state	event	next state
Idle	if not FETCH_DATA	Idle
Idle	FETCH_DATA	FetchingData
FetchingData	DATA_READY	Publishing
FetchingData	FETCH_FAILURE	Error
Publishing	PUBLISH_SUCCESS	Idle
Publishing	PUBLISH_FAILURE	Error
Error	RESET	Idle

The action λ

The **next-output function** $\lambda : Q \times \Sigma \to \Sigma$: We use $\Lambda$for the set of *next-output* functions, and $\Lambda = \{\lambda_1, \lambda_2, \lambda_3, \lambda_4\}$, where

- λ_1 = startFetch: Initiates the data fetching process, leading to either a successful fetch (σ_2) or a failure (σ_3).
- λ_2 = startPublish: Initiates the data publishing process, resulting in either a successful publish (σ_4) or a failure (σ_5).
- λ_3 = errorHandler: Handles errors by emitting a reset event (σ_6) to revert the system to the Idle state.

```
<span class="table-number">Table 3:</span>
  The next-output function $\lambda$ for producer actor - a symbolic definition:
```

current state	event	$\lambda \in \Lambda$	output
s_1	σ_1	λ_1	σ_2 or σ_3
s_2	σ_2	λ_2	σ_4 or σ_5
s_3	σ_3	λ_3	σ_6
s_3	σ_5	λ_3	σ_6

```
<span class="table-number">Table 4:</span>
  Descriptive Output Function $\lambda $
```

Current State	Input Event	Action	Output
FetchingData	FETCH_DATA	startFetch	DATA_READY / FETCH_FAILURE
Publishing	DATA_READY	startPublish	PUBLISH_SUCCESS / PUBLISH_FAILURE
Error	FETCH_FAILURE	errorHandler	RESET
Error	PUBLISH_FAILURE	errorHandler	RESET

Diagram

Simulation

The transition function δ is extended inductively into $\bar{\delta} : Q \times \Sigma^* \to Q$ to describe the machine's behavior when fed whole input words. For the empty string ε, $\bar{\delta}(q, \varepsilon) = q$ for all states q, and for strings wa where a is the last symbol and w is the (possibly empty) rest of the string, $\bar{\delta}(q, wa) = \delta(\bar{\delta}(q, w), a)$. The output function λ may be extended similarly into $\bar{\lambda}(q, w)$, which gives the complete output of the machine when run on word w from state q.

This allows the following to be defined:

Accepting word

A word $w = a_1 a_2, \ldots a_n \in \Sigma^*$ is an accepting word for the automaton if $\bar{\delta}(q_0, w) \in F$, that is, if after consuming the whole string w the machine is in an accept state.

Recognized language

The language $L \subseteq \Sigma^*$ *recognized* by a machine is the set of all the words that are accepted by the machine, $L = \{w \in \Sigma^* \mid \bar{\delta}(q_0, w) \in F\}$.

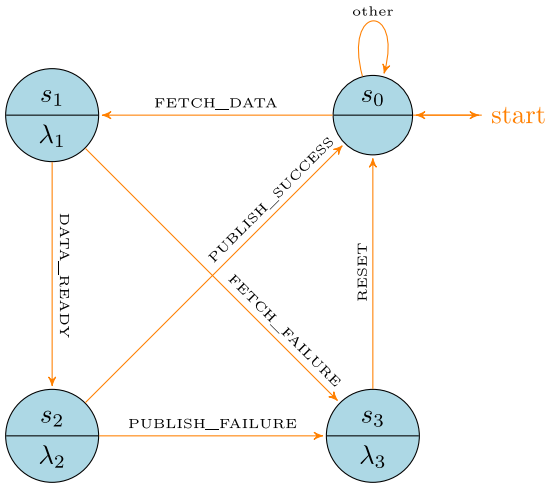

Figure 1: diagram

Simulation 1: Successful Data Fetch and Publish

- **Input Sequence:** $w = \sigma_1\sigma_2\sigma_4$
- **Step-by-Step Execution:**
  1. **Initial State:** s_0 (Idle)
  2. **Event** σ_1 (FETCH_DATA):
     - Transition: $s_0 \to s_1$
     - Action: startFetch
     - Output: σ_2 (DATA_READY)
  3. **Event** σ_2 (DATA_READY):
     - Transition: $s_1 \to s_2$
     - Action: completeFetch
     - Output: σ_4 (PUBLISH_SUCCESS)
  4. **Event** σ_4 (PUBLISH_SUCCESS):
     - Transition: $s_2 \to s_0$
     - Action: completePublish
     - Output: Idle state confirmed
  - **Final State:** s_0 (Idle) is an accepting state.
  - **Result:** w is an accepting word.

Simulation 2: Data Fetch Failure

- **Input Sequence:** $w' = \sigma_1\sigma_3\sigma_6$
- **Step-by-Step Execution:**
  1. **Initial State:** s_0 (Idle)
  2. **Event** σ_1 (FETCH_DATA):
     - Transition: $s_0 \to s_1$
     - Action: startFetch
     - Output: σ_2 (DATA_READY) or σ_3 (FETCH_FAILURE)
  3. **Event** $ _3 $ (FETCH_FAILURE):
     - Transition: $s_1 \to s_3$
     - Action: handleFetchError
     - Output: σ_6 (RESET)
  4. **Event** σ_6 (RESET):
     - Transition: $s_3 \to s_0$
     - Action: resetFromError
     - Output: σ_6 (RESET)
  - **Final State:** s_0 (Idle) is an accepting state.
  - **Result:** w' is an accepting word despite the failure, indicating a potential need to revisit acceptance

criteria.

Simulation 3: Multiple Fetch and Publish Cycles with an Error

- **Input Sequence:** $w'' = \sigma_1\sigma_2\sigma_1\sigma_3\sigma_6$

  Step-by-Step Execution:

  1. s_0 (Idle) receives σ_1 (FETCH_DATA):
     - Transition: $s_0 \to s_1$
     - Action: λ_1 (startFetch)
     - Output: Emits σ_2 (DATA_READY)
  2. s_1 (FetchingData) receives σ_2 (DATA_READY):
     - Transition: $s_1 \to s_2$
     - Action: λ_2 (startPublish)
     - Output: Emits σ_4 (PUBLISH_SUCCESS)
  3. s_2 (Publishing) receives σ_1 (FETCH_DATA):
     - Per Table 1: σ_1 is not σ_4 or σ_5 for s_2, so assuming s_2 remains to handle current publish, or define additional transitions.
     - If undefined: Assuming other events lead to state staying or ignoring.
  4. s_1 (FetchingData) receives σ_3 (FETCH_FAILURE):
     - Transition: $s_1 \to s_3$
     - Action: λ_3 (errorHandler)
     - Output: Emits σ_6 (RESET)
  5. s_3 (Error) receives σ_6 (RESET):
     - Transition: $s_3 \to s_0$
     - Action: λ_6 (resetErrors)
     - Output: Emits no further events.

  - **Final State:** s_0 (Idle), an accepting state.

  Result: w'' is an accepting word, even though a fetch failure occurred, again highlighting the need to reconsider acceptance criteria.