**Concepts & Definitions**

Event-driven systems work by [responding to Events](https://www.state-machine.com/qpc/srs_evt.html). In general, the system's response to a given Event depends both on the nature of that Event (captured in its [Signal](https://www.state-machine.com/qpc/srs_evt.html#srs_evt-sig)) and on the *history of events* the system has received.

In practice not all aspects of the full "history of past events" are relevant. The simplified history consisting only of aspects that are consequential for the system's response to *future events* is called the **Relevant History**.

**State**

**State** is an *equivalence class* of past histories of a system, all of which are equivalent in the sense that the future behavior of the system given any of these past histories will be identical. Thus, the concept of "State" is the most efficient representation of the Relevant History of the system. It is the minimum information that captures only the relevant aspects for the future behavior and abstracts away all irrelevant aspects.

**Transition**

**Transition** is a change from one State to another during the lifetime of a system. In event-driven systems, a change from one state to another can be caused only by an event. The events that triggers a Transition is called **Triggering Event** or just **Trigger** of the Transition.

**State Machine**

**State Machine** is the set of all States (*equivalence classes* of relevant histories), plus all the Transitoins (rules for changing States). An important benefit of the State Machine formalizm is the expressive graphical representation of State Machines in form of **state diagrams**.

Note

This definition pertains to event-driven State Machines, which is the only kind supported in QP Framework. The definition does not

**Hierarchical State Machine**

**Hierarchical State Machine** (a.k.a. UML statechart) is an advanced formalism which extends the traditional state machines in several ways. The most important innovation of UML state machines over classical state machines is the introduction of **hierarchically nested states**. The value of state nesting lies in avoiding repetitions, which are inevitable in the traditional “flat” state machine formalism. The semantics of state nesting allow substates to define only the *differences* in behavior from the superstates, thus promoting sharing and reuse of behavior.

**State Machine Implementation Strategy**

State Machines, and Hierarchical State Machines, in particular, can be implemented in many different ways. A specific way of implementing a state machine will be called here a **State Machine Implementation Strategy**, and it can be characterized by the following properties:

* efficiency in time (CPU cycles)
* efficiency in data space (RAM footprint)
* efficiency in code space (ROM footprint)
* monolithic vs. partitioned with various levels of granularity
* maintainability (with manual coding)
* maintainability (via automatic code generation)
* *traceability* from design (e.g., state diagram) to code
* *traceability* from code back to design
* other, quality attributes (non-functional requirements)

No single State Machine Implementation Strategy can be optimal for all circumstances, and therefore QP Framework shall support multiple and interchangeable strategies (see [REQ-QP-02\_20](https://www.state-machine.com/qpc/srs_sm.html#REQ-QP-02_20)).

**Dispatching Events to a State Machine in QP Framework**

The event processing inside a state machine is called **dispatching** an event to the state machine, and it requires interaction between the QP Framework and the QP Application:

A diagram of a machine

Description automatically generated

*Figure 02\_01:Event Dispatching to a State Machine in QP Framework*

**State Machine Specification**

The "State Machine Specification" is provided inside the QP Application and is prepared according to the rules defined by the chosen [State Machine Implementation Strategy](https://www.state-machine.com/qpc/srs_sm.html#srs_sm-impl) in QP Framework. Typically an implementation strategy represents a state machine as several elements, such as states, transitions, etc.

The "State Machine Specification" can mean state machine code (when the state machine is coded manually) or a state machine model (when the state machine is specified in a modeling tool, like ["QM"](https://www.state-machine.com/products/qm)). Either way, it is highly recommended to *think* of the state machine implementation as the **specification** of state machine elements, not merely code. This notion of "specifying" a state machine rather than coding it can be reinforced by selecting an expressive and fully *traceable* state machine implementation strategy, see [REQ-QP-02\_40](https://www.state-machine.com/qpc/srs_sm.html#REQ-QP-02_40). The advantage of a traceable implementation is that each artifact at all levels of abstraction (design to code) unambiguously represents an element of a state machine.

**State Machine Processor**

A state machine is executed in QP Framework by the "State Machine Processor" that decides which elements of the "State Machine Specification" to call. Once called, the chosen part of the "State Machine Specification" executes some actions and *returns* back to the "State Machine Processor" (QP Framework) with the status information as to what has happened. For example, the returned status might inform the "State Machine Processor" that a state transition needs to be taken, or that the event needs to be propagated to the superstate in the hierarchical state machine.

**Run To Completion (RTC) Processing**

The "State Machine Processor" is a *passive* software component that needs to be explicitly called from some control thread to dispatch each event to the given state machine object. The most important restriction is that the dispatch operation must necessarily run to completion (**Run-to-Completion** processing) before another event can be dispatched to the same state machine object.

Note

RTC event processing means, among others, that a state machine should **NOT** block or busy-poll for events (e.g., a semaphore-wait or busy-delay) because every such blocking or busy-polling call represents waiting for an *event*, which will be delivered immediately after the call unblocks. The problem is that such a "backdoor" event is delivered before the original RTC step completes, thus violating the RTC semantics. Blocking inside a state machine also extends the RTC processing and makes the state machine unresponsive to new events.