EMBSYS 110, Spring 2017 Week 4

- Week 4
  - UML (carried over from Week 2).
  - QP internals and framework design.
- References
  - [1] Samek, Miro. Practical Statecharts in C/C++, 2nd Edition. Newnes. 2009.
  - [2] Quantum Leaps, LLC. Application Note QP and ARM Cortex-M. 2016.

    http://www.state-machine.com/doc/AN_QP_and_ARM-Cortex-M.pdf

UML

- UML stands for Unified Modeling Language.
- It is a graphical language. We need some drawing tools, such as Visio, MagicDraw, StarUML, UMLet, etc.
- UML supports many different types of diagrams. These are most common and useful:
  - Class Diagram.
  - Sequence Diagram.
  - Statechart.

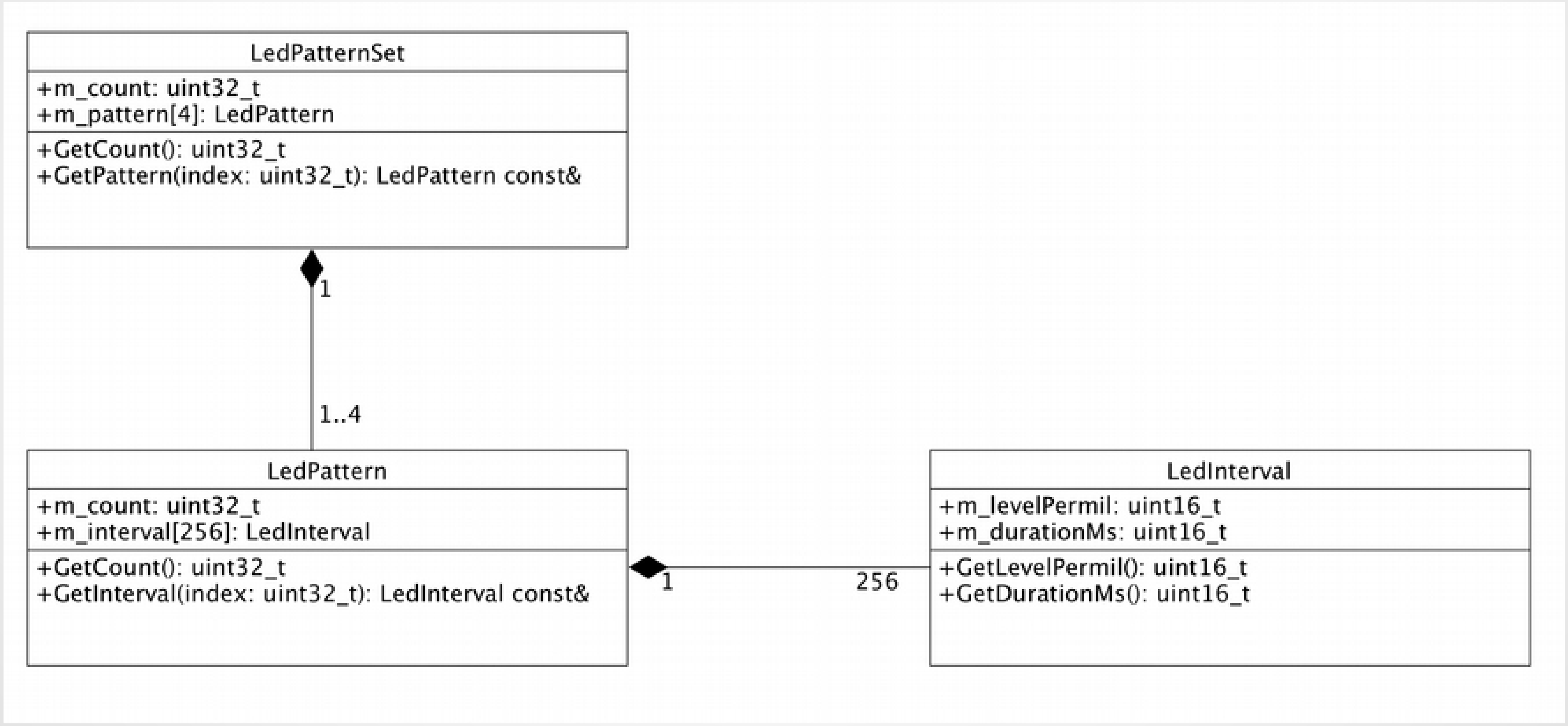
UML Class Diagram

- Class diagram.
  - Shows internal structure of a class by listing its data members.
  - Shows functional API of a class by listing its member fucntions.
  - Use ‘+’, ‘-’ and ‘#’ to show visibility of a member.

    (public, private and protected).
  - Shows static relationship between classes, such as association, aggregation and inheritance.
  - Does not show behaviors of a class, such as what a member function does or how a variable is used.

UML Class Diagram

- Example of class diagram:



LedPatternSet
+m_count: uint32_t
+m_pattern[4]: LedPattern
+GetCount(): uint32_t
+GetPattern(index: uint32_t): LedPattern const&
1
1..4
LedPattern
+m_count: uint32_t
+m_interval[256]: LedInterval
+GetCount(): uint32_t
+GetInterval(index: uint32_t): LedInterval const&
1
256
LedInterval
+m_levelPermil: uint16_t
+m_durationMs: uint16_t
+GetLevelPermil(): uint16_t
+GetDurationMs(): uint16_t

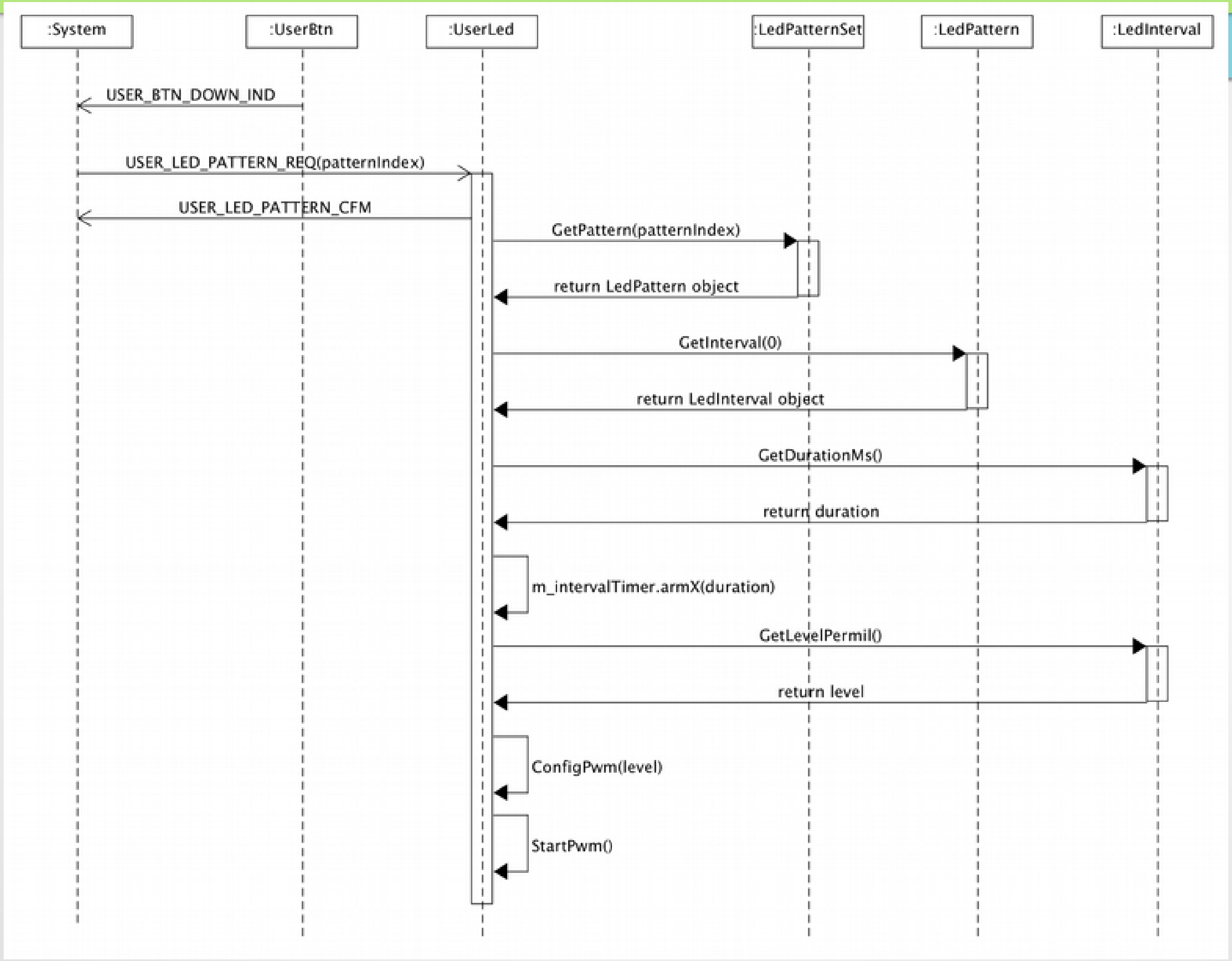
UML Sequence Diagram

- Sequence diagram.
  - Shows the interaction between two of more objects.
  - Interaction can be synchronous (function calls) or asynchronous (events/messages).
  - Each sequence diagram only shows one main scenario with a limited set of alternate paths. It usually shows one success case and/or certain failure paths. However it does not show *all* possible paths.
  - Does not show what happens if the order of messages is different or a failure occurs at a different place.
  - A metaphor – it only shows a cross section of a 3D object.

UML Sequence Diagram

- You may need an ~infinite number of sequence diagrams to completely represent all possible scenarios.
- One may be tempted to start coding based on one sequence diagram, usually a success case and worry about failure cases later.
- Good for analysis, but hard to show complete design.
- See Real-Time Software Design for Embedded Systems book Chapter 9 for details.

UML Sequence Diagram

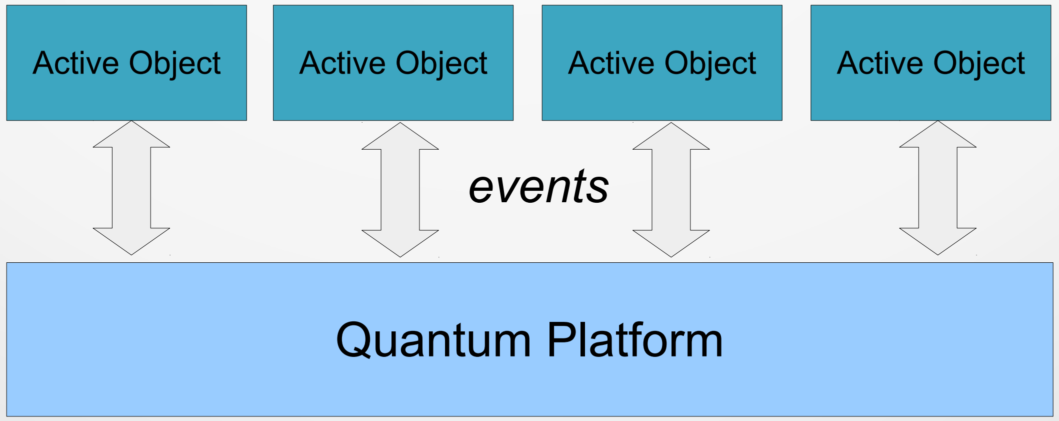


UML Statechart

- Statechart
  - Shows the *complete* behaviors of an object.
  - Complete, precise and concise.
  - Unless we design the statechart upfront and map (or translate) it to code, the state behaviors will be implicit and it will be hard to extract them from source code.
- Statechart may look complex, but it doesn't add complexity. It just reveal it.

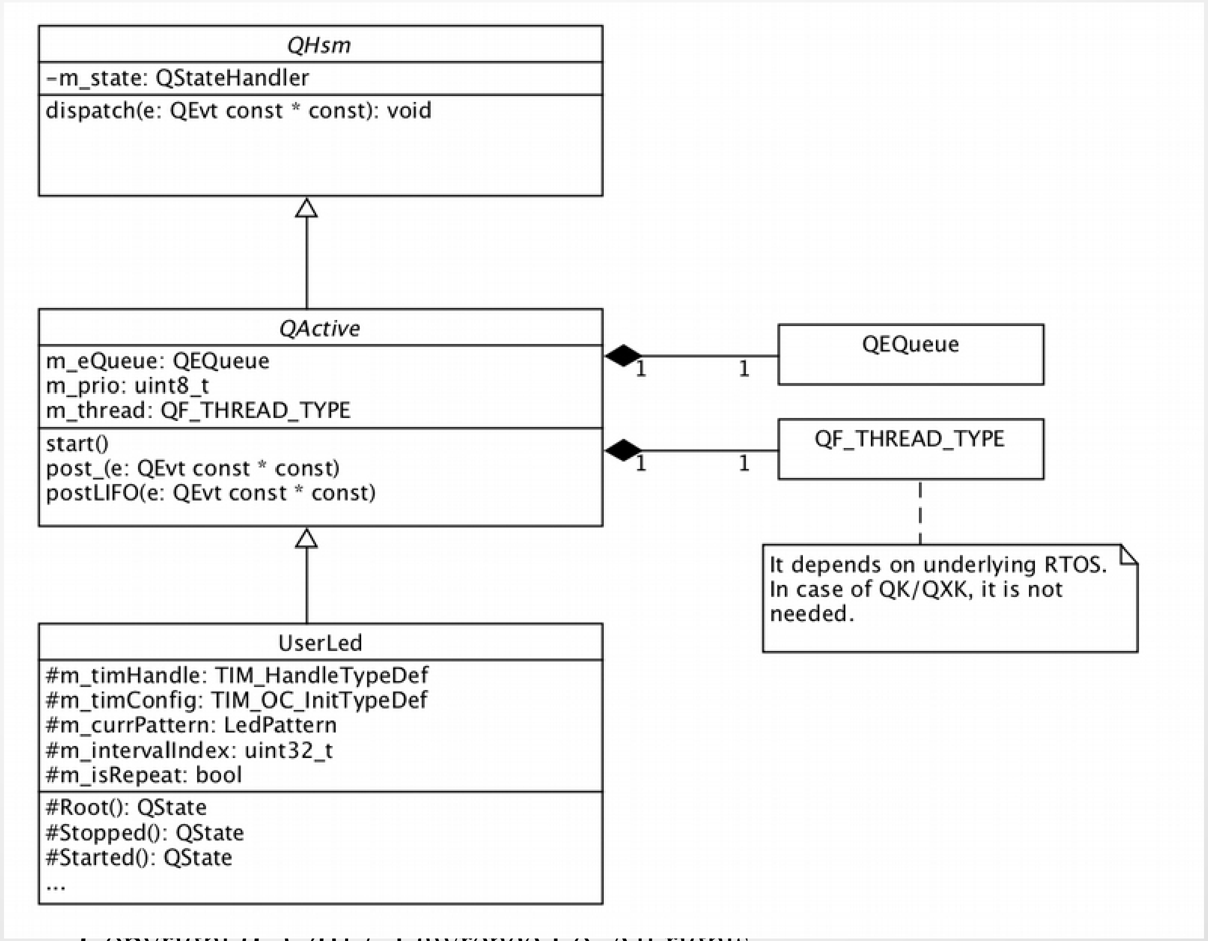
The Need for Framework

- A statechart framework allows us to focus on designing system behaviors, rather than the underlying support which is reinvented for every project.
- An example of such framework is Quantum Platform.
- The system is composed of *concurrent communicating state machines* (via events).
- With QP the architecture looks simple and neat:



Active Object

- Now that we know what inheritance is, and we have seen concrete examples of active objects, we can look at it in more details:

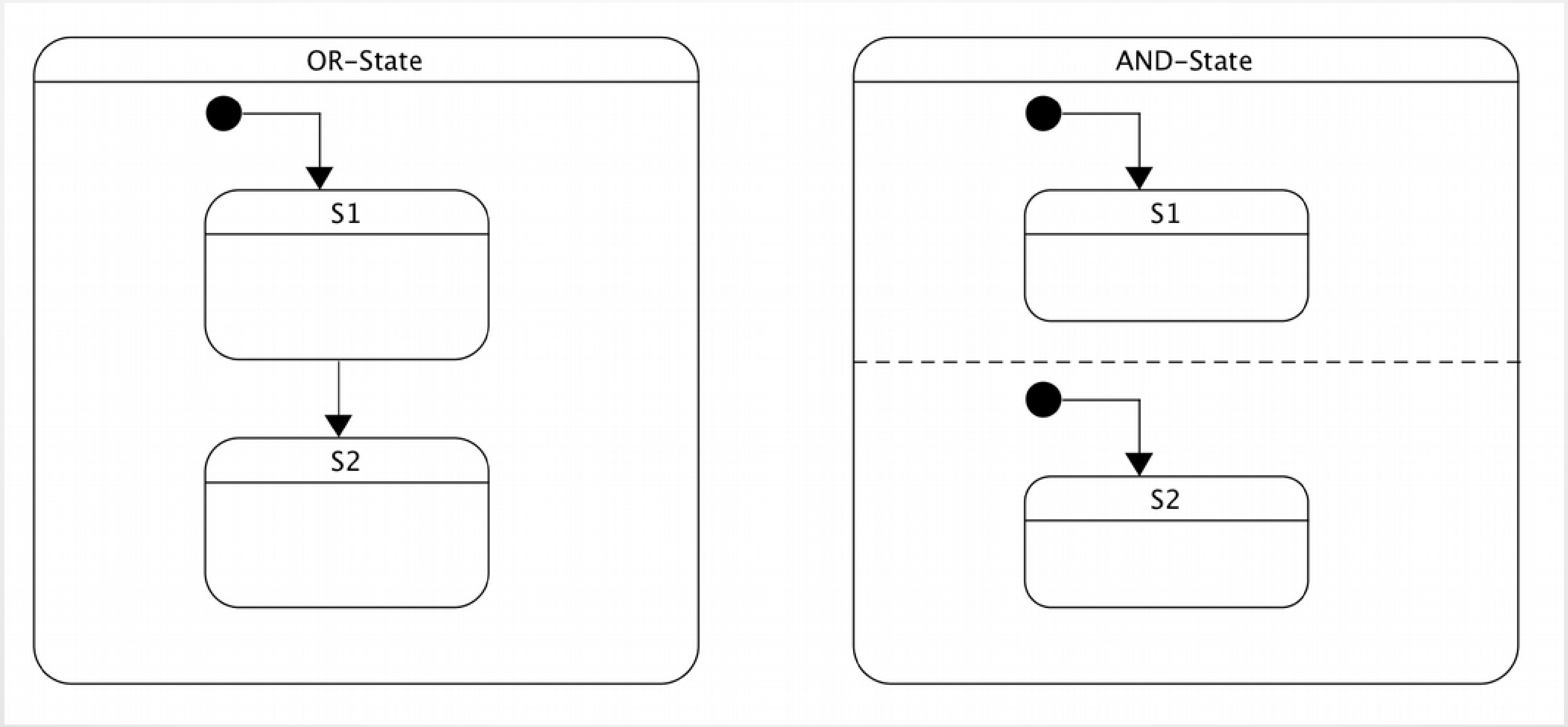

QHsm
-m_state: QStateHandler
dispatch(e: QEvt const * const): void
QActive
m_eQueue: QEQueue
m_prio: uint8_t
m_thread: QF_THREAD_TYPE
start()
post_(e: QEvt const * const)
postLIFO(e: QEvt const * const)
QEQueue
QF_THREAD_TYPE
1
1
1
1
It depends on underlying RTOS. In case of QK/QXK, it is not needed.
UserLed
#m_timHandle: TIM_HandleTypeDef
#m_timConfig: TIM_OC_InitTypeDef
#m_currPattern: LedPattern
#m_intervalIndex: uint32_t
#m_isRepeat: bool
#Root(): QState
#Stopped(): QState
#Started(): QState
...

Active Object

- An active object is a hierarchical state machine (QHsm) with an event queue and a thread/task.
- In contrast to a passive object that exposes a functional API (public member functions) to be called by a user object, an active object *actively* gets events from its event queue to process.
- This promotes decoupling among objects – they don't need to know each other by *direct reference* in order to call each other's member functions.
- In contrast to a free thread (an ordinary RTOS thread) it enforces explicit state-based behavoral modeling.

Orthogonal Region

- According to David Harel's original idea, orthogonal regions are parallel states within a composite state (called an AND-state).

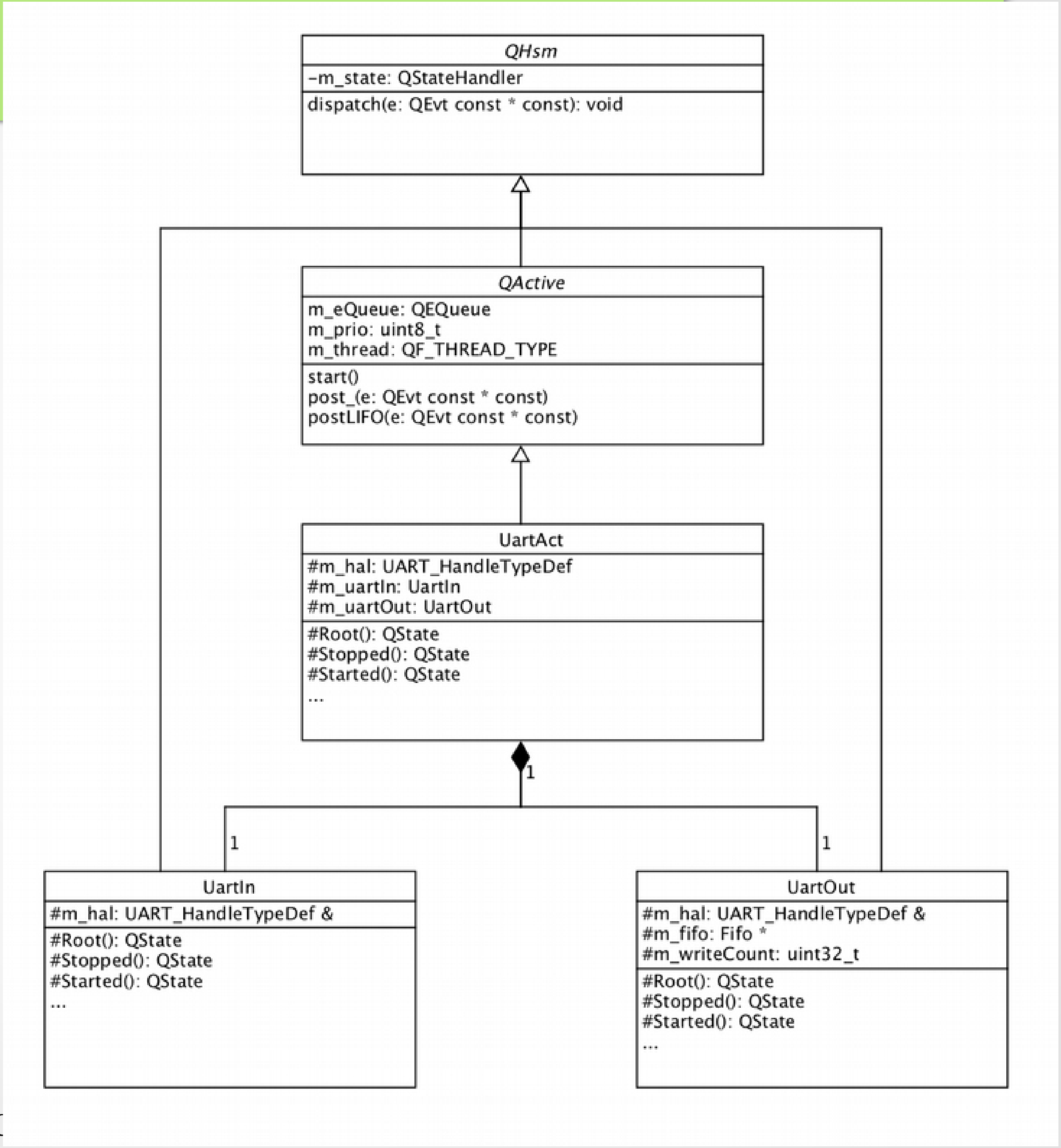


Orthogonal Region

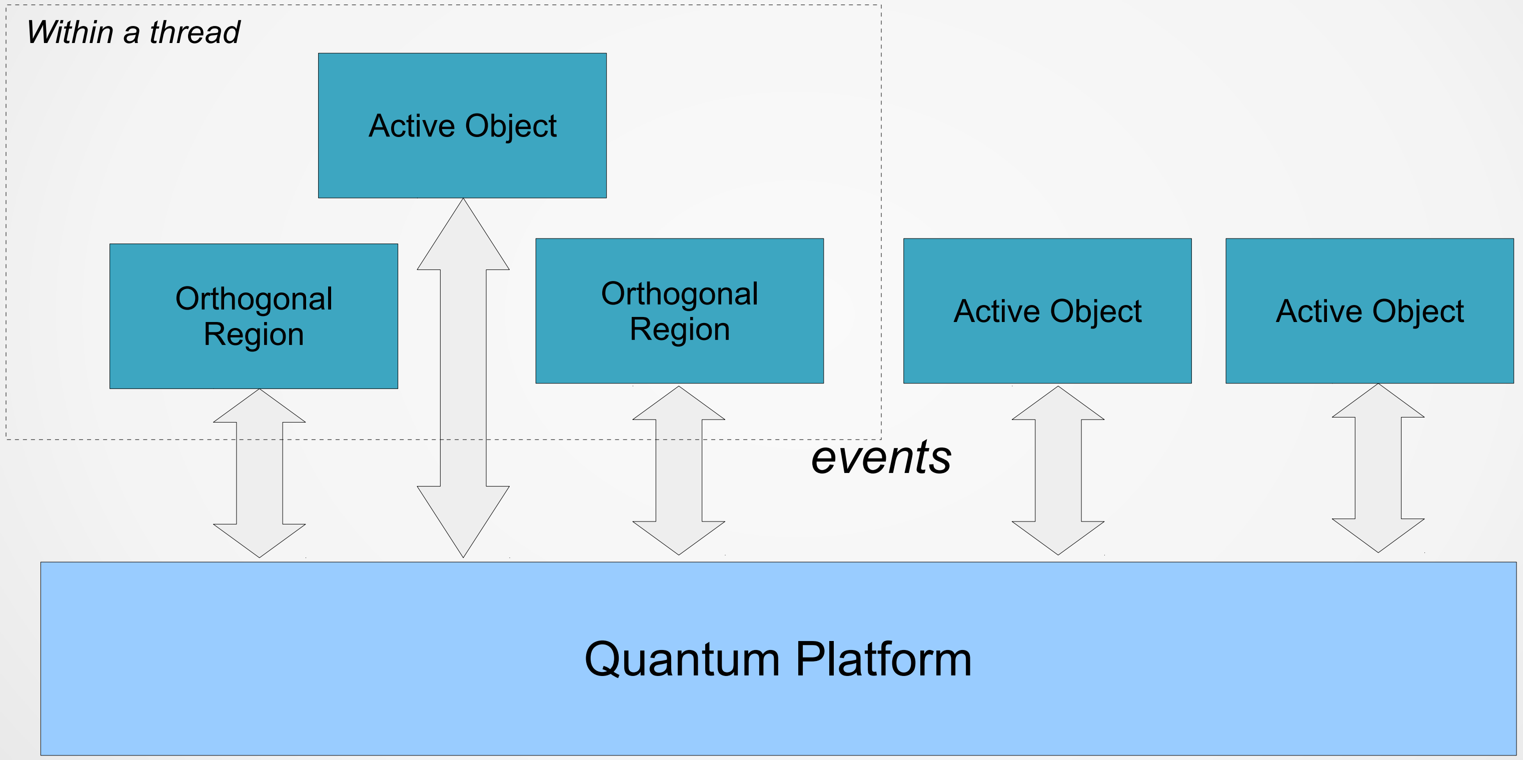
- Orthogonal regions behave like parallel state machines within the containing state machine.
- There is a subtle difference between “orthogonal regions” and “concurrent state machines”.
- QP does not support orthogonal regions inherently. It indireclty supports it via an application design pattern using aggregation.
- In QP, you can think of orthogonal regions as parallel state machines within an active object (all sharing the same thread and event queue) (though they may not be as closely related as in Harel’s definition.)

O.R.

- Example of orthogonal regions:

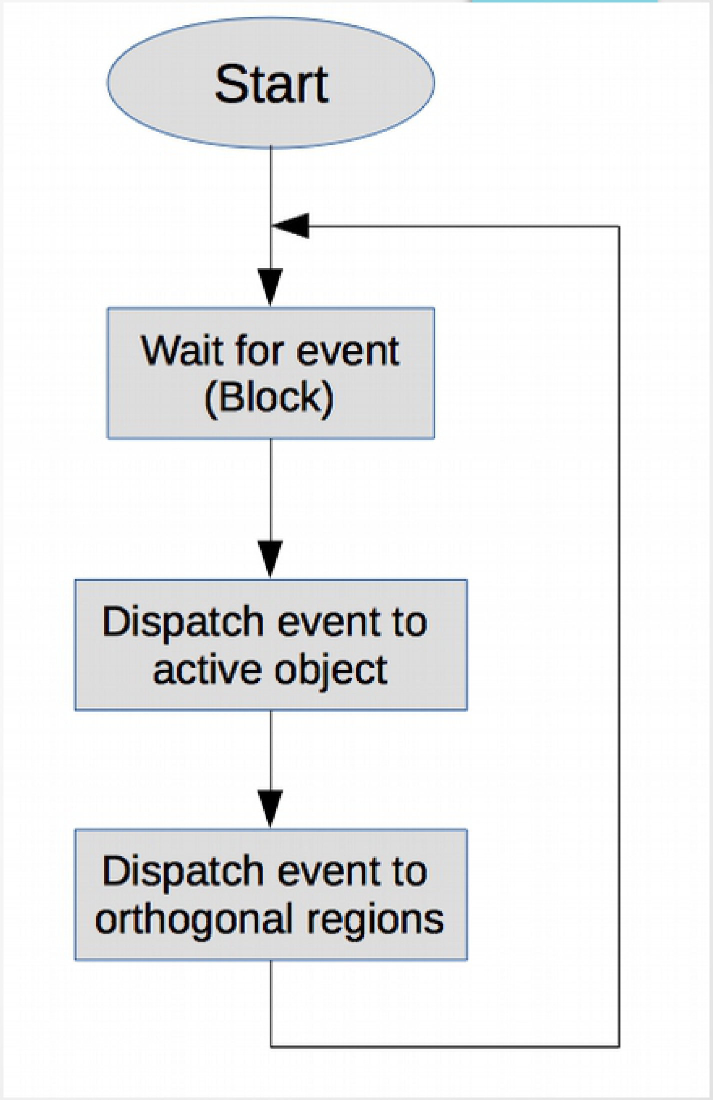


O.R.


Within a thread
Active Object
Orthogonal Region
Orthogonal Region
Active Object
Active Object
events
Quantum Platform

Execution Model

- The diagram on the right shows the execution flow of *one* active object.
- There is a single blocking point at its own event queue.
- Active objects communicate via unified mechanism (event exchange).
- Behaviors of each active object are precisely specified by statecharts.
- Indirect support of orthogonal region via object aggregation.


Start
Wait for event (Block)
Dispatch event to active object
Dispatch event to orthogonal regions

Execution Model

- Terminology – Microstep and Macrostep.

  (See Andrzej W¡sowski)
- Microstep
  - The processing of a *single* event retrieved from the event queue of an active object.
- Macrostep
  - A chain of microsteps triggered by a single external event.
  - A microstep can generate an internal event posted to its own event queue (via postLIFO()).
  - The above step can reiterate.
  - A macrostep ends when there are no more internal events generated. At this point, the state-machine has reached stability as a result of the initiating external event.
  - Note we *MUST* use **postLIFO()** to post an internal event to the front of the event queue to ensure it is processed immediately as the next event, such that there will be no other external events interrupting the current macrostep; otherwise it may cause race conditions.

QP Internals

- The core of QP is an “event processor” (QEP)
  - Like any ordinary state machines, it dispatches an event to the state-handler function of the current state.
  - It does much more than that since it supports state hierarchy.
- On top of the event processor, it provide basic/standard services for an event-driven framework, such as:
  - Event pools (qmpool.h, qf_mem.cpp, qf_dyn.cpp).
  - Event queues (qequeue.h, qf_qeq.cpp, qf_qact.cpp).
  - Timers (qf.h, qf_time.cpp).
  - Publish-subscribe event routing (qf.h, qf_ps.cpp).
- QP can run on top of a traditional RTOS such as UCOS-II or in Linux. It can also run on top of its own real-time kernel (QV, QK or QXK).

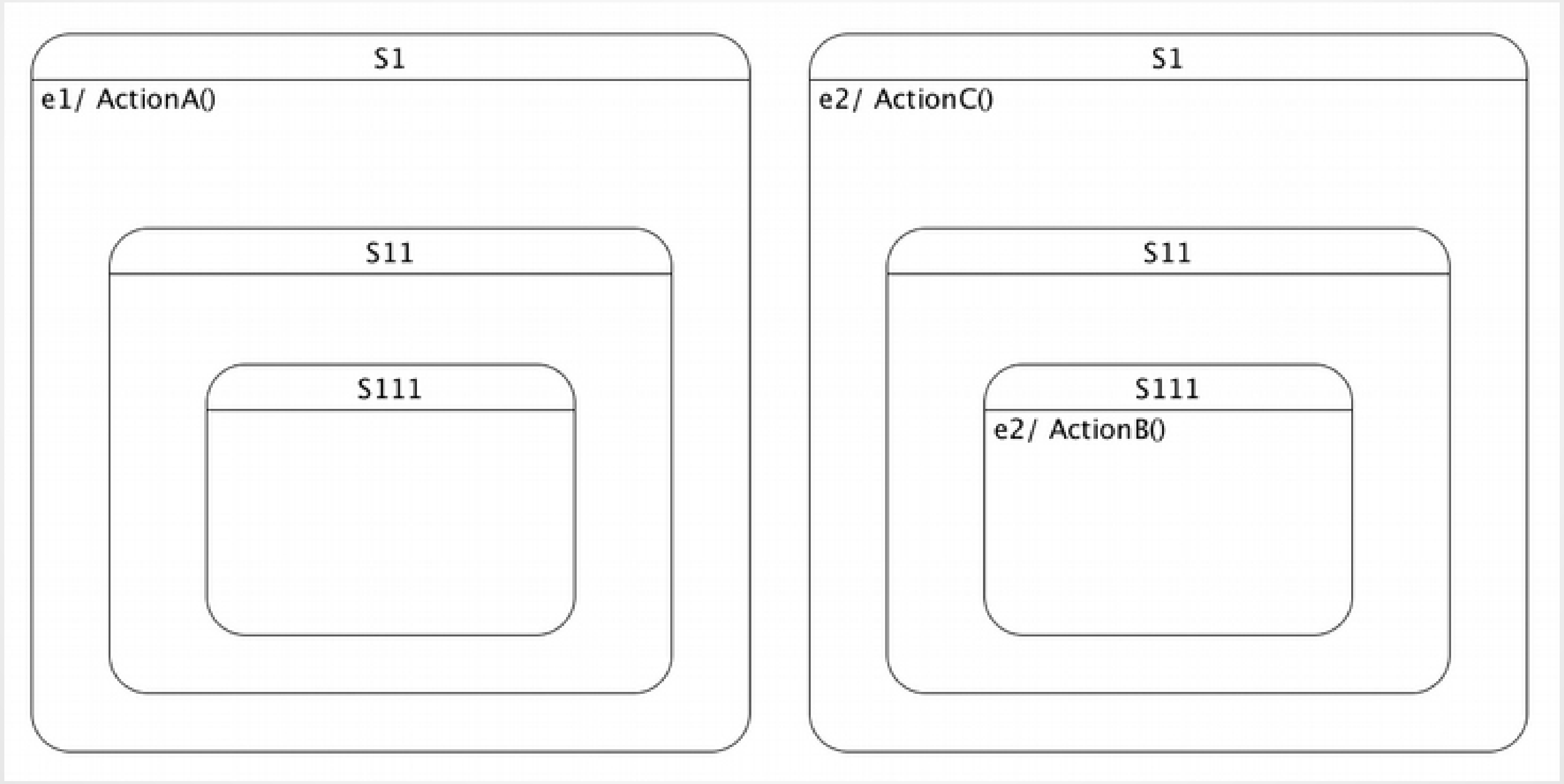
Event Processor (QEP)

- The core algorithm is implemented in Qhsm::dispatch() in qep_hsm.cpp.
- The code is hard to read (optimized). Main ideas are explained below.
- Statechart semantics is rather complicated. Check-out UML 2.0 Spec. It is quite informal. It can be ambiguous since it uses natural language. Just released a beta version of the “Precise Semantics Of UML State Machines Version 1.0 – Beta 1”. It is fresh, dated 2017 April 1 (after our class has started!)
- For precise description of statechart semantics, it requires formal language. See reference of Andrzej W¡sowski.
- For most practical uses, it is sufficient to understand the semantics actually used by the tools, which is QP in our case.

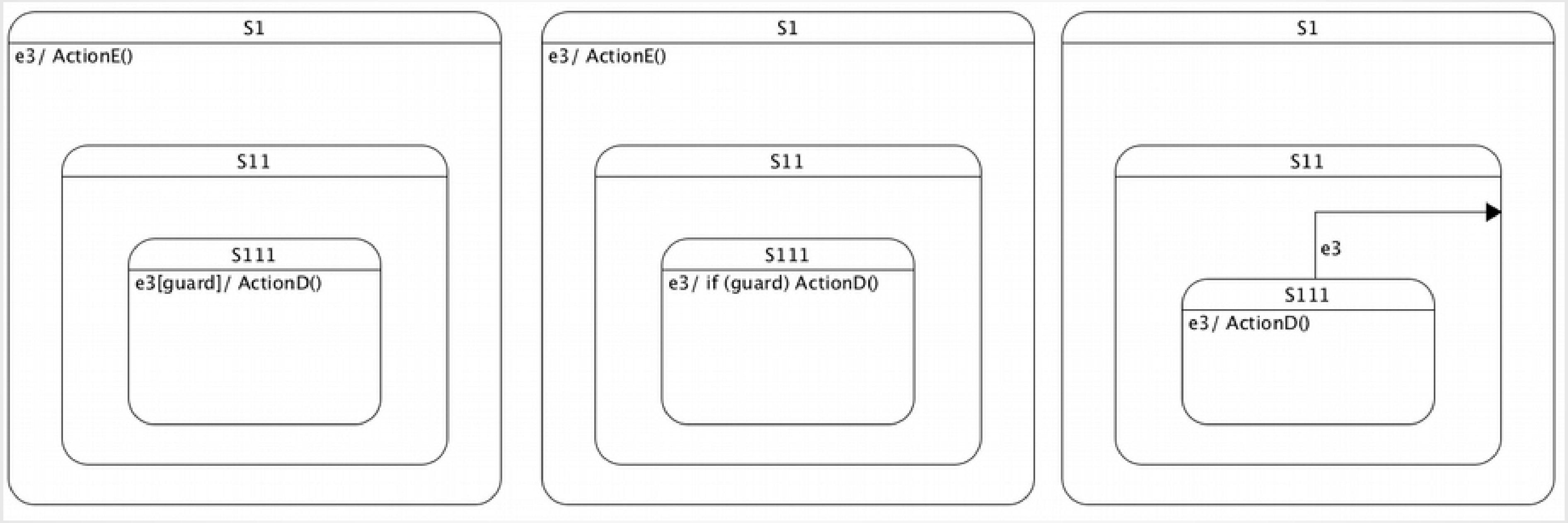
QEP

- The first concept to understand is the hierarchical dispatching of events.
- Transitions are evaluated to match an incoming (triggering) event type from the current leaf state (most nested) outward to the top state.
- Similar to function overriding of subclass in C++.
- In other words, priority is given to more nested state.
- A statechart is ill-formed if there are more than one matches (enabled transition) at a certain state (nesting level).
- Note how a guard is different in semantics than an ‘if’ statement.
- Note in the following examples we mainly use internal transitions for illustration but the same principle applies to state transitions.

QEP



QEP

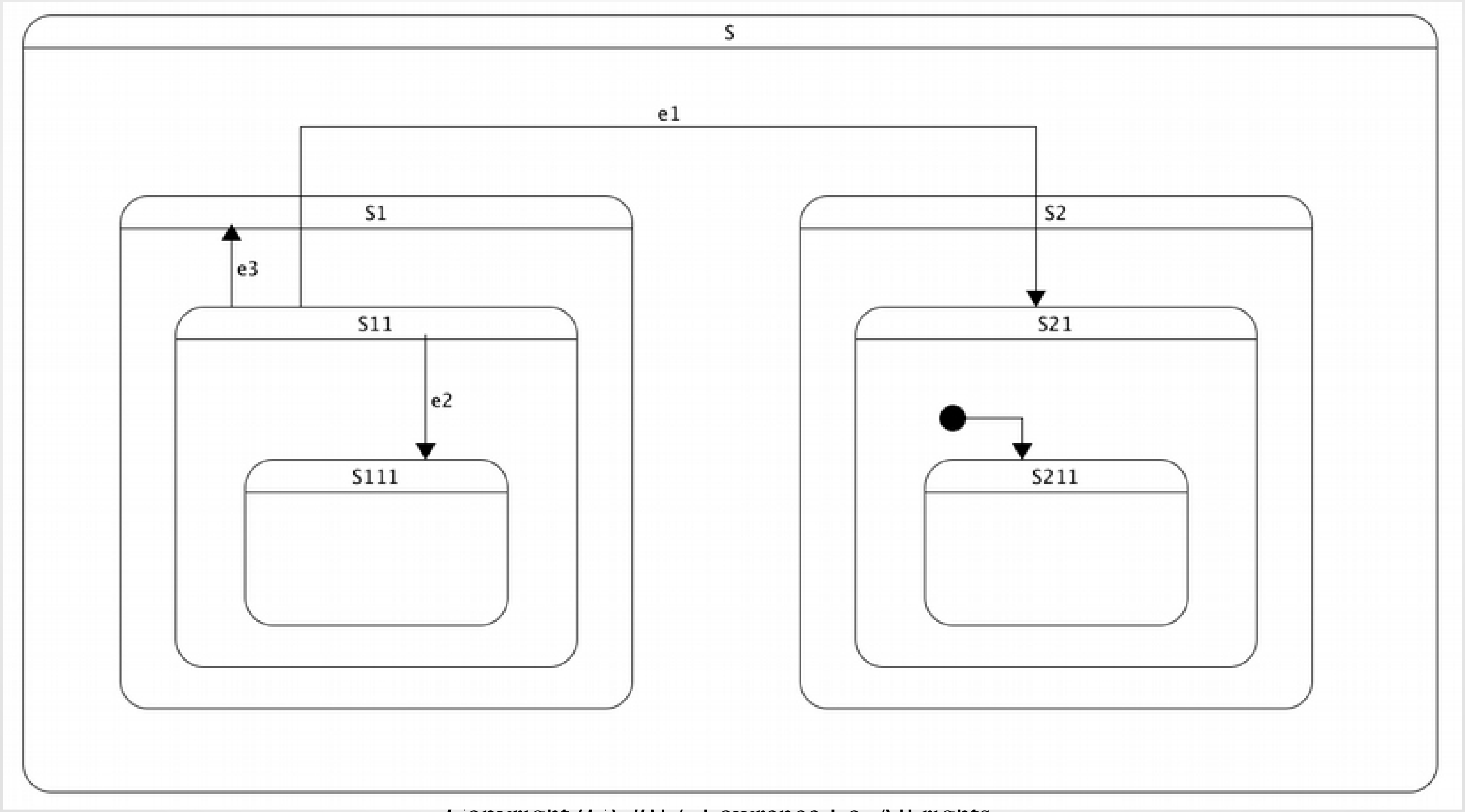

S1
e3/ ActionE()
S11
S111
e3[guard]/ ActionD()
S1
e3/ ActionE()
S11
S111
e3/ if (guard) ActionD()
S1
S11
e3
S111
e3/ ActionD()

QEP

- The second important concept is to understand the *three* parts of a state transition.
- Understand the terminology:
  - Current state (current leaf state, most-nested)
  - Transition source (originating state of a transition line)
  - Transition target (terminating state of a transition line)
  - Final state (new “current” leaf state after transition)
- Transition source can be the same as current state, or a parent of the current state.
- Final state can be the same as transition target, or a child of the transition target.
- Note when the state machine is in a parent state (OR-state), it MUST be in one and only one of its immediate child states.
- The terms composite state, parent state, super state are interchangable, so are child and substate.

QEP

- Example illustrating typical transition types (current state S111).



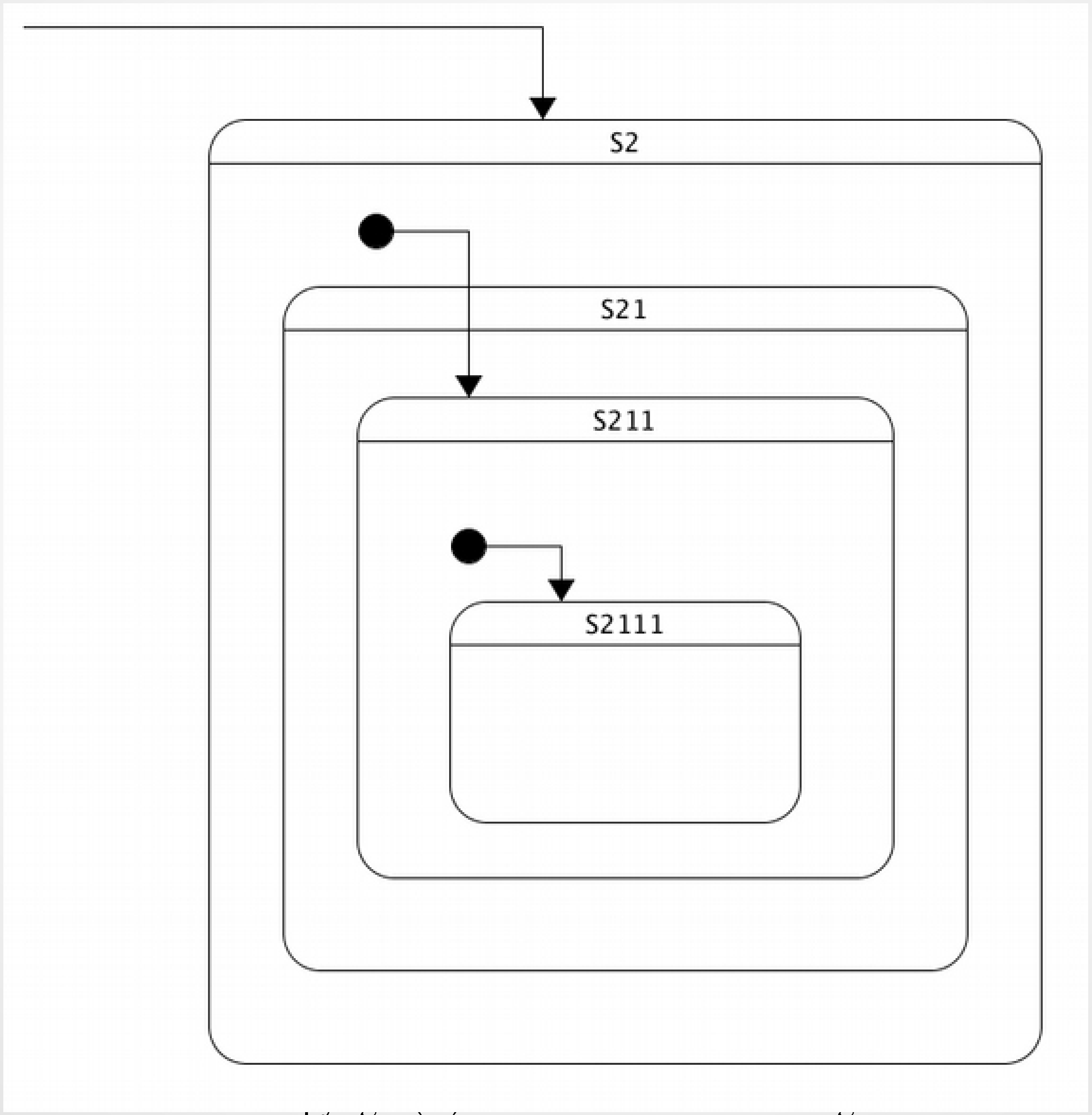
QEP

- Part 1
  - Exit actions from current state to transition source.
    - NOT exit transition source.
    - If both are the same, nothing is done.
- Part 2
  - Exit actions from transition source to (but not including) lowest common ancestor (LCA).
    - Call hsm_tran() to exit to LCA.
  - Entry actions from (but not including) LCA to transition target.
    - Back-tracing path[ ] set up by hsm_tran().
  - See PSCiCC Figure 4.6 (page 178) for all transition categories to consider in hsm_tran(). Will explain shortly.
  - In general, there are three kinds of state transitions. Look for e1, e2 and e3 in previous figure. ‘e2’ and ‘e3’ are special cases in which the LCA is either the transition source or transition target respectively.
  - Literally it is not strictly correct since one cannot be an ancestor of itself. But it is easier to think it that way for the cases of e2 and e3. As stated previously, it does NOT exit and re-enter the LCA.
  - In UML 2, the above transition semantics is called “local transition” (see PSCiCC Section 2.3.9 page 81). The alternative is called “external transition” which is NOT supported by QP.
  - Let’s read Section 2.3.9 together to confirm we understand the subtle difference.

QEP

- Part 3
  - Entry actions from transition target to a leaf state.
    - Once reaching the transition target, if it is not a leaf state already, it needs to *recursively* execute initial transitions until a leaf state is reached.
    - Note that an initial transition may cross multiple state layers (though uncommon). All entry actions are executed in the path of an initial transition.
    - Recall the rule stated earlier: *When the state machine is in a parent state (OR-state), it MUST be in one and only one of its immediate child states.* This part of a transition satisfies this rule.
    - Illustration shown next...

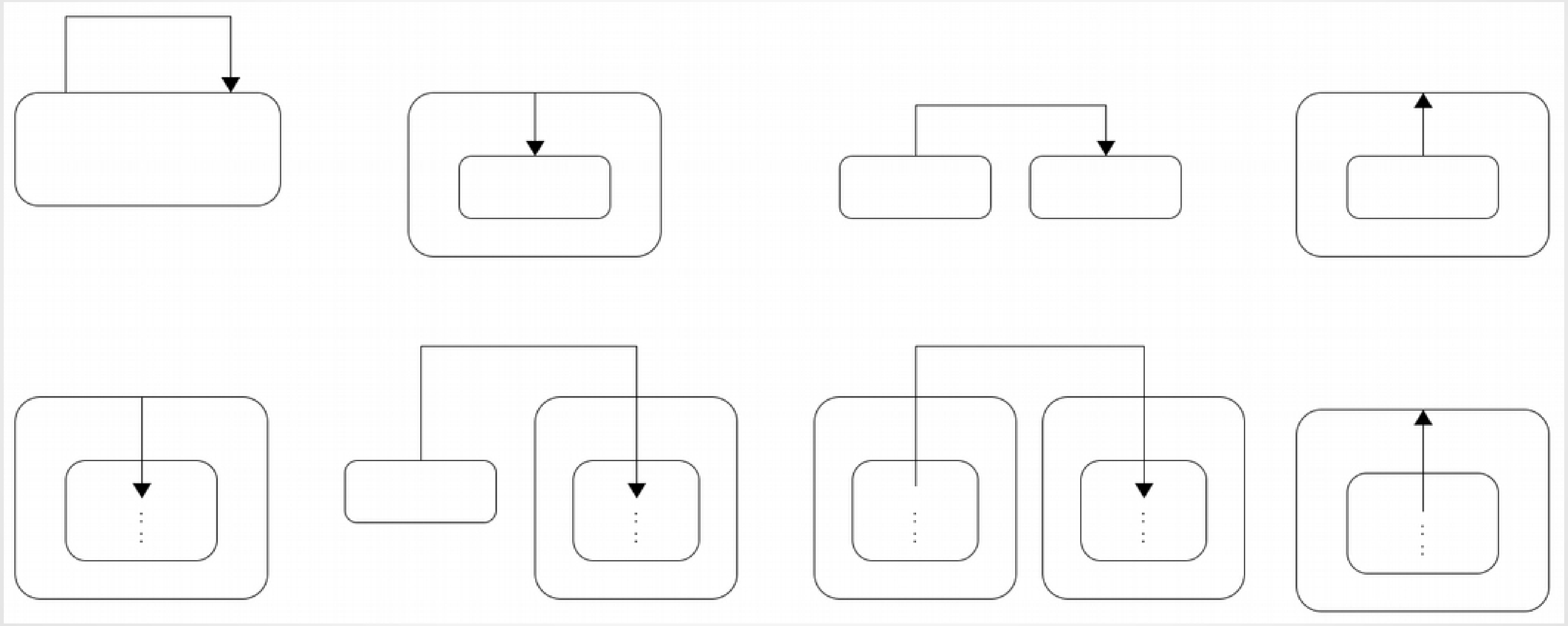
QEP



QEP

- Let’s take a closer look at hsm_tran().
- Recall that in Part 2 of a transition, it does the following:
  - *Exit actions from transition source to (but not including) lowest common ancestor (LCA).*
    - *Call hsm_tran() to exit to LCA.*
  - *Entry actions from (but not including) LCA to transition target.*
    - *Back-tracing path[ ] set up by hsm_tran().*
- QP optimizes the common cases and makes special cases for them to make them efficient (at the expense of code size and readability).
- There are 8 cases shown on the next page. Note they still fall into the 3 categories we have seen before (e1, e2 and e3).

QEP



QEP Summary

- Summary of QHsm::dispatch():
  - Propagate events to super states until handled. Conflict resolution.
  - Note the use of EMPTY_SIG. Note the difference between current state and transition source.
  - Exit actions from current state to transition source.
  - Call hsm_tran() to exit to LCA (Lowest common ancestor).
  - Upon return from hsm_tran(), execute entry actions from LCA to transition target. (Back tracing path[ ]).
  - Once reached transition target, recursively execute initial transition until reaching the leaf state. All entry actions are executed in the path of initial transition.

QEP Summary

- See PSCiCC Figure 2.11 (page 88) for illustration.
- Non-trivial algorithm. Well tested in open source and commercial uses.
- Optimized but there is still run-time cost for state transition. Avoid unnecessary transition.
- You can try out QM which generates code for you automatically.
  - It is faster than hand-coded approach since all transition paths are determined during code-generation.
  - However QM is freeware but not open-source. You may get locked-in.
  - You may pick your favorite IDE with our current approach.
  - While it’s a big advantage to enforce synchronization between code and statechart, our current approach gives you flexibility to decide how much detail you want to put in to a statechart. (Action/guard can be a description.)

<End of Class>

End of Class 4.

The following slides are for reference only.

Event Framework

- Should we build our application directly on top of an RTOS?
- If you do you may find yourself repeating some supporting classes over and over, such as event queues, memory pools, timers, etc.
- We need a higher level of abstraction. As introduced in Q2, an event-driven framework is one common choice.
- Without an event framework, we would think of free-running threads (each with arbitrary structures) interacting with heterogeneous IPC mechanisms.

QXK

- QXK is the *extension* of QK (Quantum Kernel).
- QXK is a preemptive RTOS, just like UCOS-II.
- There are two types of threads:
  - Basic threads (a.k.a. AO thread).
  - Extended threads (can block).
- The original QK only supports basic threads which *cannot* block. Each thread is an active object that processes events in run-to-completion steps.
- Being non-blocking means that it is truly asynchronous.

QXK

- Note on asynchronous design:
  - Asynchronous vs synchronous.
  - Callback vs events.
- Being non-blocking means the highest priority task (i.e. thread/active object) always run to completion.
- Simplifies context-switching design, allowing the use of a single stack (main stack) vs one for each task (process stack).
- When does context-switching occur in RTOS?

QXK

- Answer:
  - When an RTOS API is called, which makes a higher priority task *ready*.
  - When an interrupt occurs, which makes a higher priority task *ready*.
- In traditional RTOS, the ways to make a higher priority task ready include unlocking a mutex, signaling a semaphore, posting to a mailbox, setting an event flag, timer expiring, etc.
- Do you really need all these different kinds of IPC (inter-process communication)? Think about where a task may block...
- QP argues that you only need one – event queues. Single point of blocking per task. Unified IPC.
- That's what QK supports. QXK extends it by adding semaphore, etc.

QXK

- See Figure 5. block diagram in App Note [2] (page 18).
- It illustrates the 2nd case of context-switching caused by interrupts.
- For the 1st case of context-switching caused by API calls (posting events) there are no interrupts involved (PendSV or NMI), using just function calls. More efficient. See qpcpp\include\qxk.h.

QXK

```
#define QACTIVE_EQUEUE_SIGNAL_(me_) do { \
     QXK_attr_.readySet.insert((me_)->m_prio); \
     if (!QXK_ISR_CONTEXT_()) { \
         if (QXK_sched_() != static_cast<uint_fast8_t>(0)) { \
             QXK_activate_(); \
         } \
     } \
 } while (false)
```

- Take a look at QXK_sched_() and QXK_activate_() in qpcpp\source\qxk.cpp.
- Focus on basic thread to basic thread context-switch.

QXK

- For the 2nd case of context-switch caused by interrupts, see macro in qpcpp\ports\arm-cm\qxk\iar\qxk_port.h:

```
#define QXK_ISR_EXIT()  do { \
    QF_INT_DISABLE(); \
    if (QXK_sched_() != static_cast<uint_fast8_t>(0)) { \
        QXK_CONTEXT_SWITCH_(); \
    } \
    QF_INT_ENABLE(); \
} while (false)
```

- See QXK_CONTEXT_SWITCH_() in qxk_port.h. Note that all it does is to trigger the PendSV interrupt.

QXK

- Let’s look at *PendSV_Handler* in qpcpp\ports\arm-cm\qxk\iar\qxk_port.s.
- Note the different types of context-switch:
  - (1) Basic to basic thread (single stack, using MSP).
  - (2) Extended to extended threads (individual stack for each thread, using PSP).
  - (3) Basic to extended thread.
  - (4) Extended to basic thread.
- Type (2) above is just like traditional RTOS, like UCOS-II.
- Type (1) above is good for RTC active objects.

QXK

- For extended threads (type (2)), see the following functions:
  - QXK_stackInit_fill – Initialize stack to switch to a thread handler function for the first time (assumes no FP state).
  - PendSV_save_ex – Save the complete context to the current process stack (supports FP lazy stacking).
  - PendSV_restore_ex – Restore the complete context from the next process stack.
- Note hardware automatically save part of the stack frame to the current process stack. See Figure 2-3 (page 2-27) of ARM Cortex-M4 Generic User Guide.
  - Why does it only save R0-3, R12, LR, PC and xPSR?
  - Why does it need to save the rest in PendSV_save_ex?

QXK

- For basic threads (type (1)), see the following functions:
  - PendSV_activate – Fall thru from PendSV_Handler. It is very tricky that it constructs a new stack frame so that it can return to the C function QXK_activate_() to activate the next (highest priority) active object.

    Why does it NOT save the rest of the context (i.e. R4-R11, etc)? Think about APCS. They are automatically saved by the C function when they are used!
  - Thread_ret – Return to here from QXK_activate_() where there are no more higher priority active objects. Tricky how this return address is set up in the artificial stack frame above.
  - NMI_Handler – Artificial interrupt to go back to handler mode, so it can return from the original interrupt back to the preempted thread. It is tricky how it discard the NMI stack frame (in main stack) to expose the original stack frame (pushed by the original interrupt) (again in main stack).