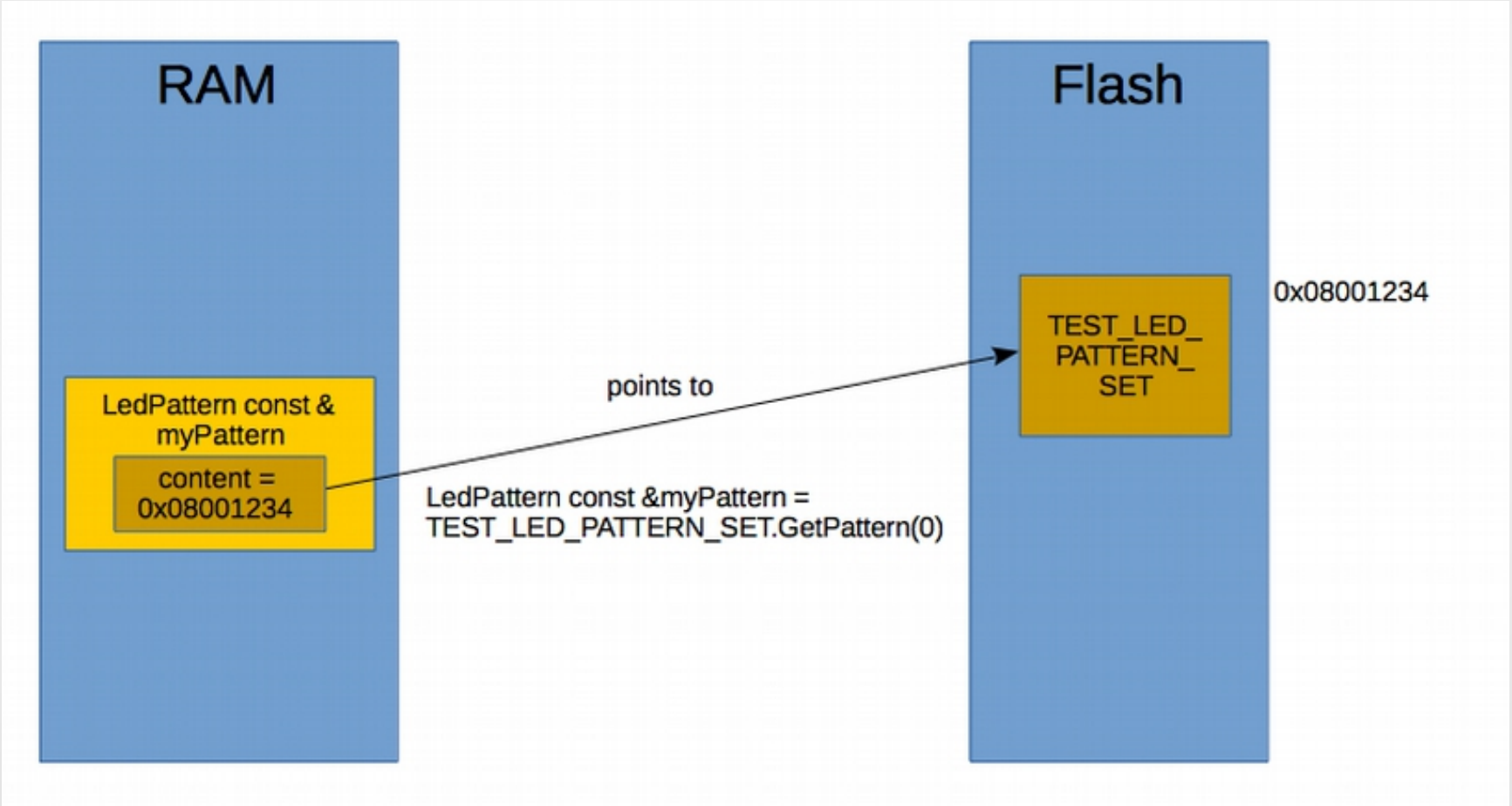
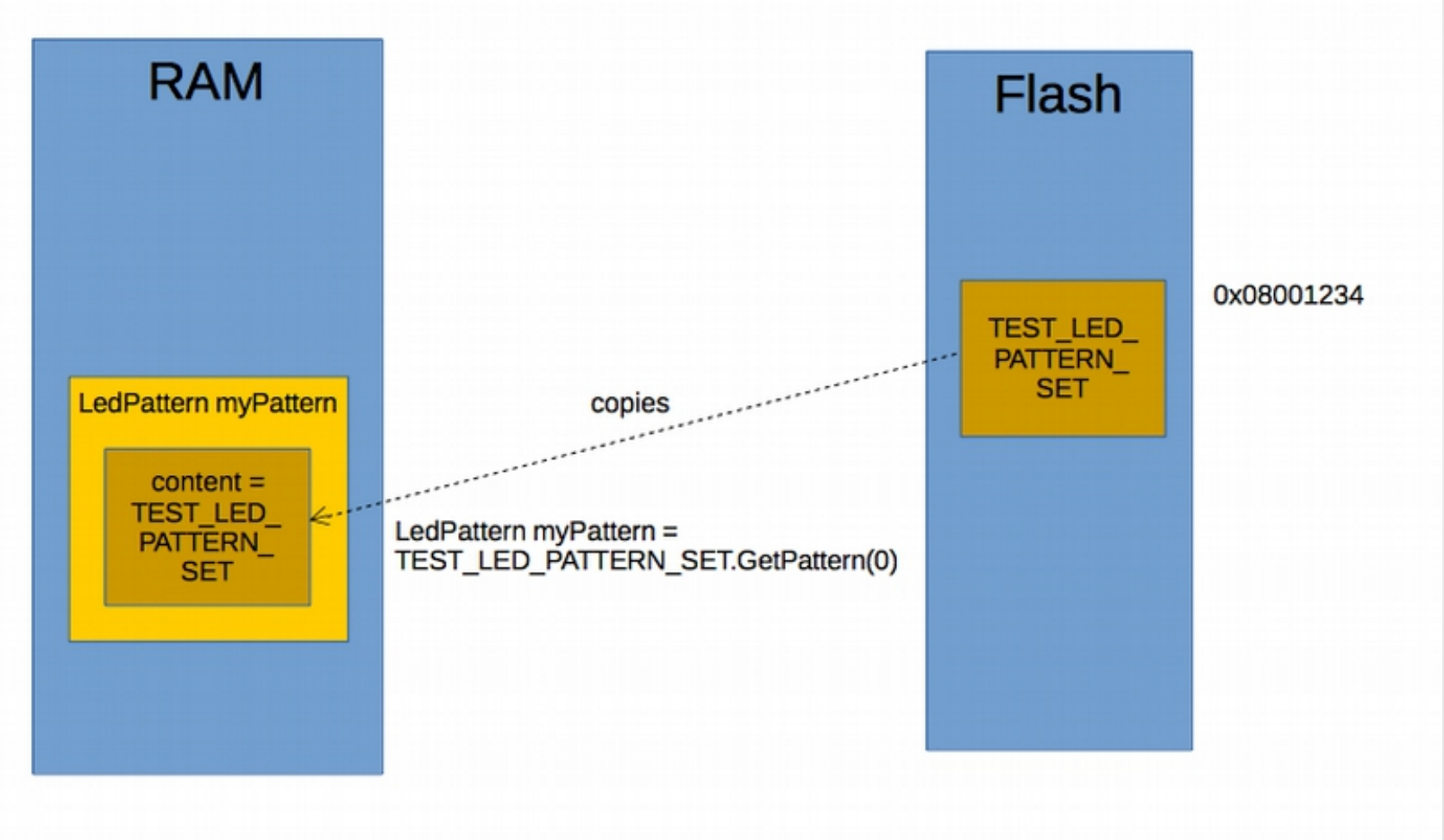
EMBSYS 110, Spring 2017 Week 5

- Week 5
  - Discussion of C++ Reference.
  - Example project (PSCiCC demo).
  - Assignment 3.
  - Design patterns and heuristics.

C++ Reference



C++ Reference



Other questions?

- In order to receive an event from another active object, you must subscribe it first.
- For example:

```
QState System::InitialPseudoState(System * const me, QEvt const * const e)
{
    ...
    me→subscribe(SYSTEM_START_REQ);
    me->subscribe(SYSTEM_STOP_REQ);
    me->subscribe(SYSTEM_STATE_TIMER);
```

Example Project

- Our IAR project is based on sample projects found in STM32 Cube package.
- An advantage is it is already customized for our target board (e.g clock configuration).
- The IAR virtual folder is closely mapped to physical directory layout. It makes it easier to locate the files if needed.
- It contains:
  - Drivers – BSP, HAL, CMSIS.
  - qpcpp – QP statechart framework.
  - Projects – Our own application.

Example Project

- Traditional C projects separate .c and .h files.
- We still keep “src” and “inc” for general source and header files.
- However for object-oriented components, it is more coherent to place .cpp and .h together.
- Let’s take a look at the main components in our example project. On top of QP, we have:
  - System – Overall system manager (active object).
  - UartAct – UART driver (active object). Contains IN and OUT regions.
  - UserBtn – User button driver (active object).
  - UserLed – User LED driver (active object).

Example Project

- Statecharts are crucial so we treat them like source. They are placed along with source files.
- Events are defined in event.h.
- ISR's are in stm32f4xx_it.cpp. Note:
  - ISR's must be declared with "extern C" (C linkage).

```
extern "C" {
    void SysTick_Handler(void);
    ….
}
```

  - They are called from *startup_stm32f401xe.s* located in Projects\MyApp\EWARM.

    (copied from Drivers\CMSIS\Device\ST\STM32F4xx\Source\Templates\iar).

Example Project

- main.c is simple. All logic/behaviors including how system starts up is modeled by statecharts and implemented in active objects (or HSM's).
- Let's do a walk-through of the statechart demo project at:

  https://github.com/galliumstudio/stm32f401-nucleo-iar-demo/tree/feature/pscicc_demo

  Refer to statechart in PSCiCC 2nd Edition Figure 2.11 on page 88.

Assignment 3

- Refer to Assignment 3 handout.

Design Patterns

- Now that we know the rules of statecharts and how to code them up (with QP) the big questions are:
  - How to partition the system into active objects/HSM's?
  - How to define the event interface?
  - How to design states of a statechart?
- In other words, given a tool we need to know how to use it effectively.
- In fact these are non-trivial questions. Even harder than the rules and implementation of statecharts themselves.

Design Patterns

- Traditional “design patterns” (Gang of Four) deals with relationship among classes. They are mostly static and structural rather than behavioral.
- They touch “state” on the surface. Check out their simplistic state pattern.
- Here we need to find patterns in
  - Partitioning of system into concurrent state machines (that communicate via events).
  - Definition and semantics of event interface.
  - Heuristics in statechart design itself.

System Partitioning

- This does not seem a hard problem, but it can be hard because
  - There are many ways of doing this. Hard to *prove* which way is better.
  - A very abstract problem. Can you ask a computer program to do it?
- One method I found very useful is called “part-whole statecharts”, or “systems of systems”.
- See:
  - Pazzi, Luca. Systems of Systems Modeled by a Hierarchical Part-Whole State-Based Formalism. March 2013.

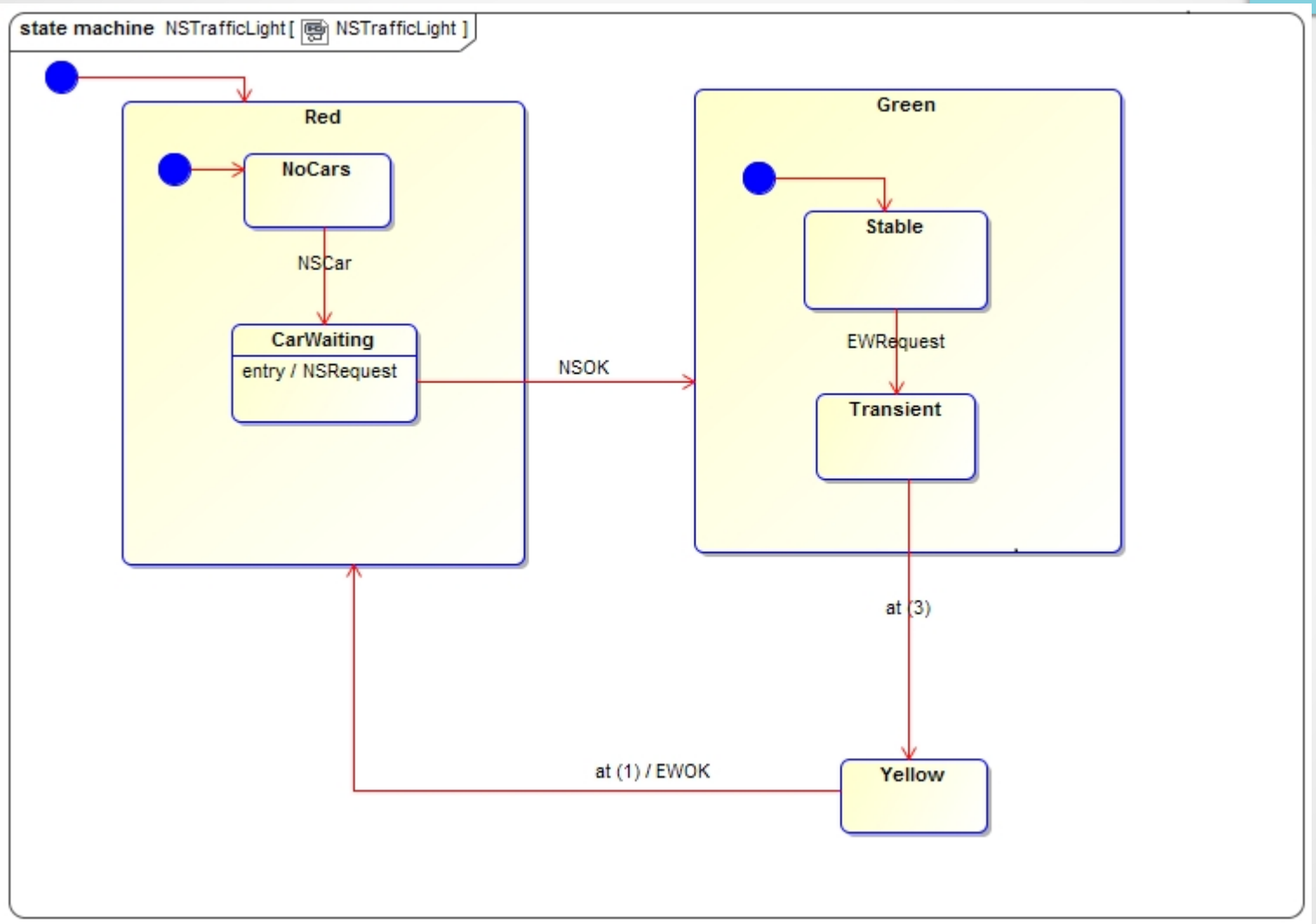
    https://www.researchgate.net/publication/236156084_Systems_of_Systems_Modeled_by_a_Hierarchical_Part-Whole_State-Based_Formalism

System Partitioning

- In a nutshell, it means recursively dividing a system into a hierarchy of components. Each component is a “whole” and a “part” at the same time.
- It is a “whole” representing lower layer components it encapsulates.
- It is a “part” to the upper layer component containing it.
- Similar to the “Coordinator Objects” in Section 8.8.2 of the Real-Time Software Design for Embedded Systems book.
- See examples in our example projects and Assignment 3.

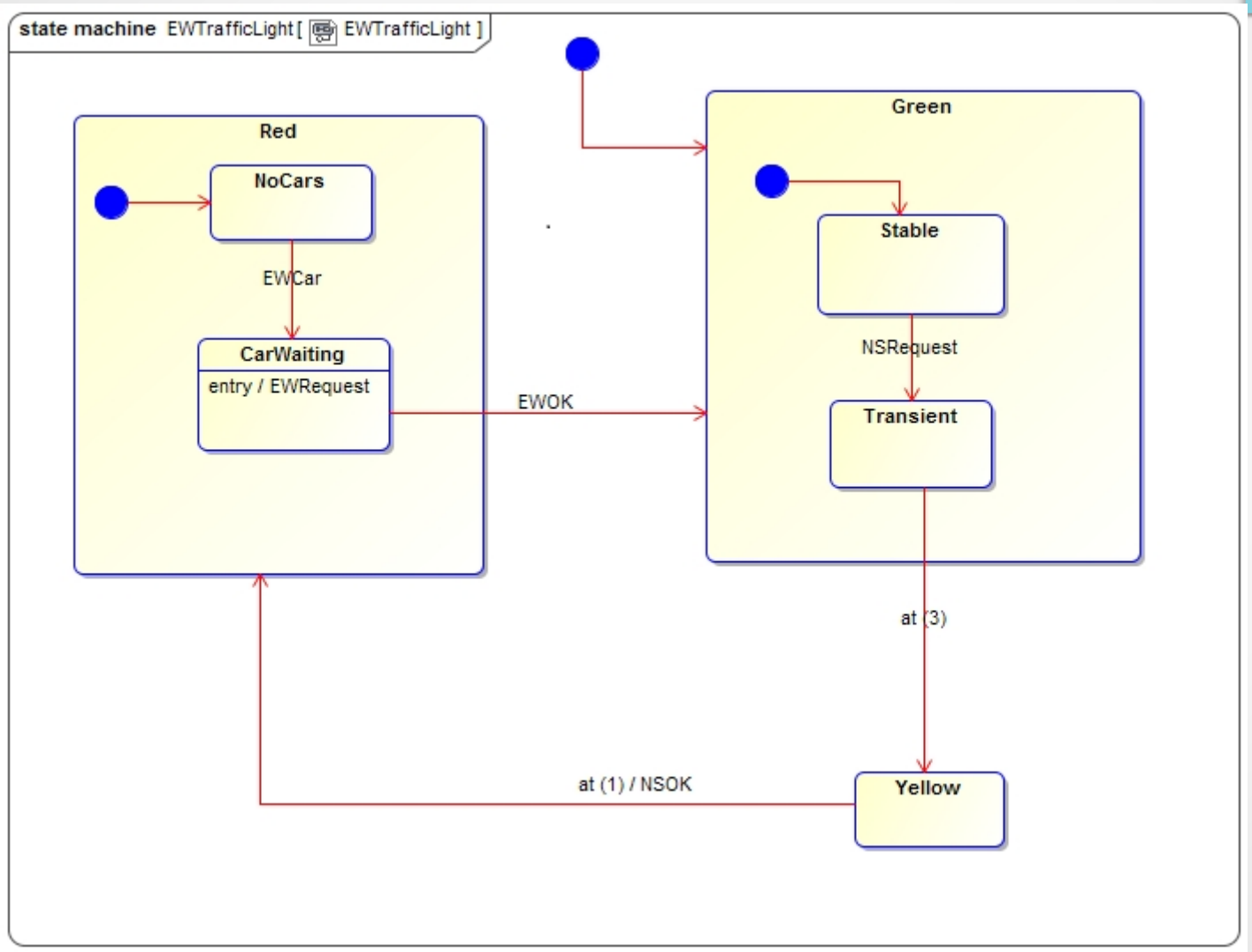
System Partitioning

https://github.com/JPLOpenSource/SCA/tree/master/autocoder/test/files/trafficlight/md16.5



System Partitioning

https://github.com/JPLOpenSource/SCA/tree/master/autocoder/test/files/trafficlight/md16.5



System Partitioning

- The concept of PWS is not unfamiliar. It is similar to the layering concept used in network protocols (7 layers OSI model).
- However it is quite novice in applying to the entire system modeled with state machines.
- Compare to the traditional modular programming paradigm using function calls. Sometimes, modules are organized in layers (or via class aggregation).
  - Similar in *asymmetry*. That is function calls only from upper to lower layers.
  - Notification from lower layer to upper layers done via callbacks.

System Partitioning

- Issues of function calls:
  - Deep and potentially recursive calls (stack overflow).
  - With multi-threading it can be hard to know in which thread/context a function is called. Think about a module API with helper thread or ISR. Data contention problem.
  - Unless using completion callback, can’t do anything else when blocked.
  - Even with completion callback, canceling an outstanding call can be tricky and error-prone (implicit or ah-hoc state machines).

System Partitioning

- Analogous to company organization. Command chain. Feedback.
- PWS simply establishes a control hierarchy among state machines (active objects or regions).
- What would happen without such hierarchy? We may end up with many state machines talking with each other at the same level. Hard to tell what the system *global state* is!
- Note – we are talking about two different type of hierarchy here, which are entirely different things.
  - One is the hierarchy of state machines in a system.
  - The other is the hierarchy of states within a single state machines (and hence the term HSM).

System Partitioning

- Note how PWS is different from orthogonal regions. Explicit vs implicit.
  - QP lacks true support for orthogonal regions.
  - The so-called *orthogonal region pattern* is QP is simply a way to encapsulate multiple HSM's into one active object or thread. Very useful if you need to support a large number of parallel HSM's (e.g. connections, clients, etc.). Simply call them "regions".
  - Question – when to create an HSM as an active object and when to put it as a region in an existing active object?

    Think about priority and synchronicity.

System Partitioning

  - True orthogonal regions have complex semantics, e.g. fork and join. Broadcast.
  - Real cases often involve *wait* so fork and join may not be practical. In other words, synchronization among true OR can be challenging. Hence explicit synchronization in PWS.
  - Better to rely on well-defined event interface among state machines for synchronization (next).
  - Literature that recommends avoiding OR altogether...
- Discuss the two safety rules in PWS (*verification* at design stage).
- Mind maps.
- More examples?

Event Interface

- QP or other event frameworks do not dictate how events are defined. It provides flexibility. Polite way of saying they don’t care.
- Easy to end up with an ad-hoc or loose set of events.
- Active objects (or in traditional RTOS terms *threads*) are asynchronous. They interact just like network nodes using network protocols.
- Events, therefore, should be defined like PDU’s of a protocols.

Event Interface

- We adopted the following common semantics:
  - REQ and CFM (request and confirmation)
  - IND and RSP (indication and response)
- They form pairs with matching sequence numbers.
  - CFM acknowledges REQ.
  - RSP acknowledges RSP.
- Question – why use sequence number? (Race conditions.)
- Acknowledgment is like returning status from a function. The norm is to check with exception allowed. (Not the other way round!)

Event Interface/Statechart Design

- Each HSM exposes an event interface. It describes the services it provides.
- How an service is fulfilled is hidden. Only care about the results. (Just like… :)
- Max timeout is part of the REQ. Otherwise how does the requesting object know how long to wait? Guaranteed to send a confirm within max timeout.
- Upon timeout, requesting object should handle the timeout exception case (as an internal event).
- Redundancy in timeout handling? Done in both requesting and requested object. Why is it necessary? Different teams work on different components. No assumption, no trust. Why does a computer program hang?

Event Interface/Statechart Design

- When designing a statechart, it is important to handle every request in every state! (So for some indications as well.)
- It sounds challenging. But thanks to state hierarchy!
- What if you missed one state? Thanks to timeout in requesting object! Now we see the importance of redundancy.
- Examples of the need to handle a request in an unexpected state? Actually a common cause for race conditions! Statechart exposes all these cases so we can see them! (People say statechart makes a design complex, but the fact is...)

Event Interface/Statechart Design

- Note the use of internal events as reminders. Useful for
  - Breaking up a long chain of actions.
  - Factorizing common actions.
  - Just to make a statechart look neater!
- Protocol reference. Q.921 LAPD/HDLC and BLE.
- Summary of statechart design pattern so far
  - Use well-defined event interface (REQ/CFM/IND/RSP).
  - Use sequence number.
  - Always check for timeout.
  - Handle all requests in all states.