Embedded systems engineering

Event-driven systems

David Kendall

Northumbria University

Introduction

- How to manage communication of external events from ISRs to tasks
- How to manage communication between tasks
- How to ensure that the system remains responsive to all events for which it is responsible.
- One disciplined approach is to use an event-driven framework
- The approach presented here follows Chapter 6 of Samek, M., Practical UML Statecharts in C/C++ Event-Driven Programming for Embedded Systems, 2nd Edition, Newnes, 2008

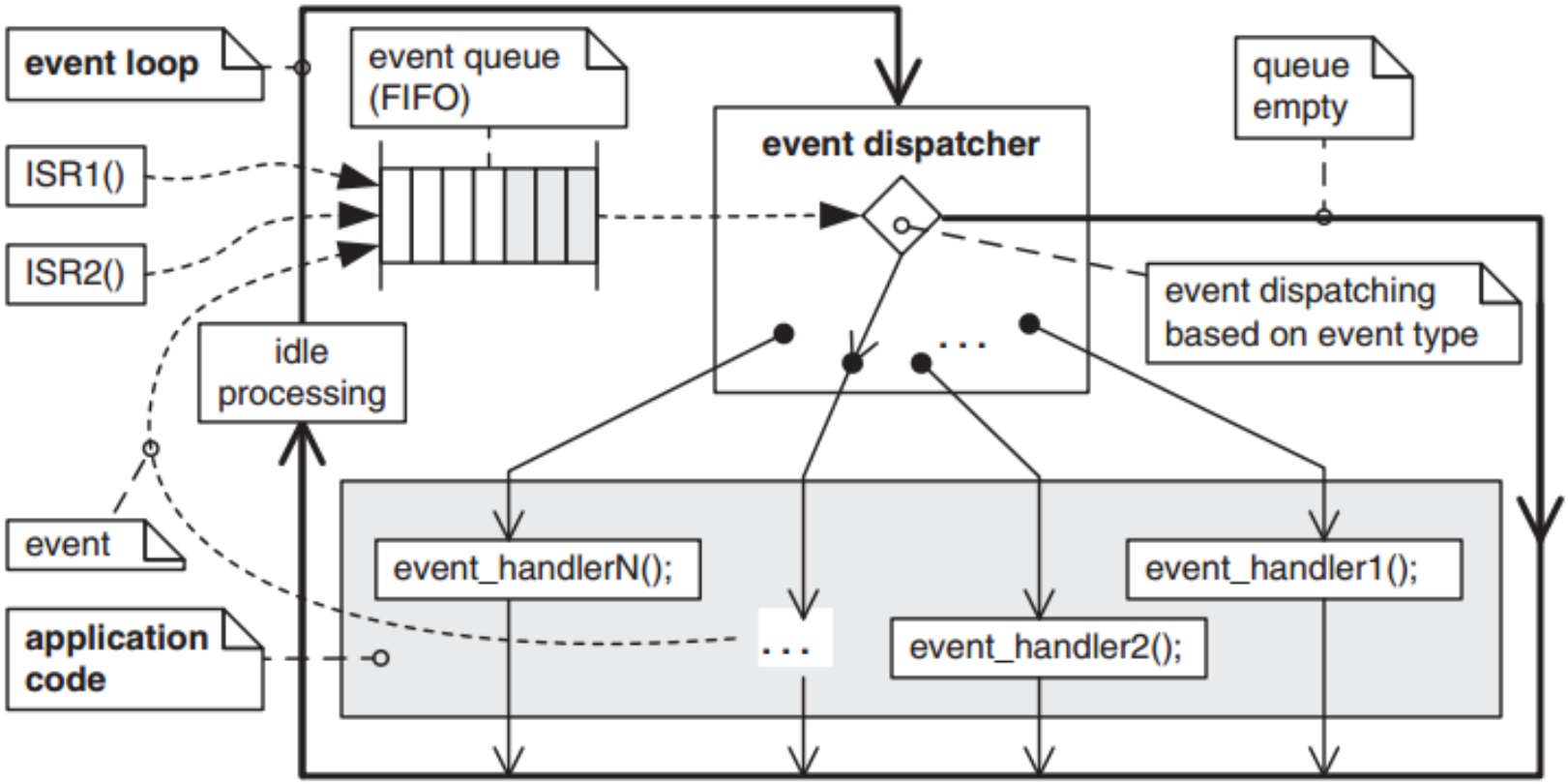
Introduction

- Samek introduces a *framework* for event-driven embedded systems that is intended to support a development approach based on *hierarchical state machines* (HSMs)
- His approach is useful *whether or not* you intend to develop with HSMs
- What you get is a *disciplined* approach to event-driven system development that produces code that is:
  - free from concurrency hazards
  - responsive
- The rest of the lecture often uses state machine examples but the approach is more generally applicable

Inversion of control

- Traditional sequential programs have a control model in which the program waits (blocks) for an event (input) whenever it needs it
  - the programmer manages the control code, but
  - the program is *unresponsive* to other events while waiting
- Traditional approach to *multi-tasking* was developed so that programmers could maintain the blocking approach to I/O with which they were familiar, while allowing the program to remain responsive
  - while one or more tasks are waiting for I/O, run some other task that is ready
  - Disadvantage: task overheads: one stack per task, context switch etc.
- Event-driven systems *invert the control*
  - Hollywood principle - “don’t call us, we’ll call you”
  - the event-driven system manages the control code
  - the programmer writes the event handlers

Traditional event-driven system



(Samek, 2008, p.264)

Traditional event-driven system (TEDS)

- The traditional event-driven system is clearly divided into
  - the event-driven infrastructure
    - event loop
    - event dispatcher
    - event queue
  - the application
    - event handlers
- Events can be generated by ISRs or by event handlers
- All events go into the event queue
- Event dispatcher is in control of the system
  - remove event from event queue
  - dispatch to appropriate event handler based on the *event type*
- Event handler - simple, non-blocking, sequential code that runs to completion and returns to the dispatcher as soon as possible
- Widely-used approach in GUI frameworks, e.g. MFC, X-Windows, Swing, Qt etc.

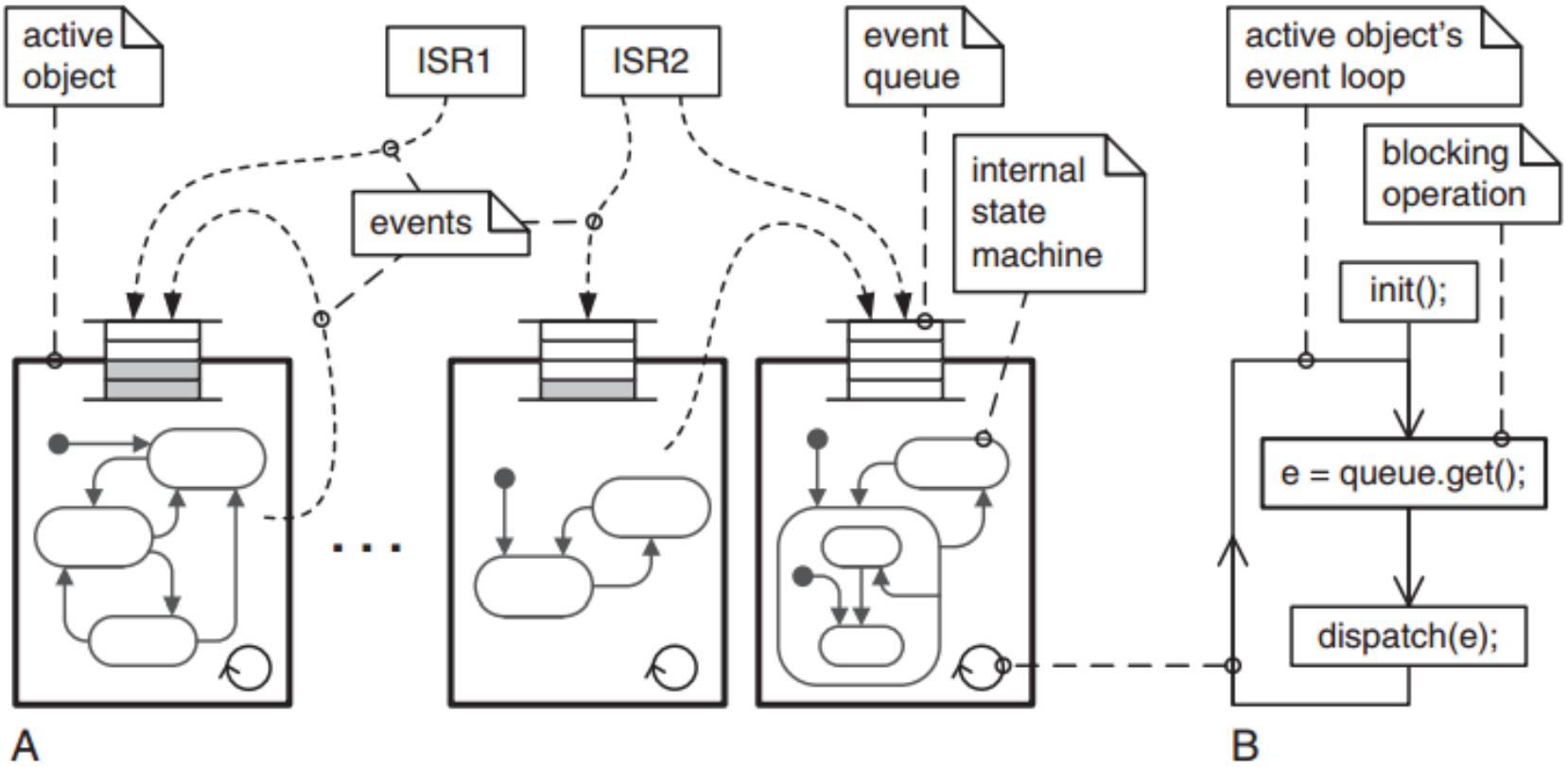
Characteristics of TEDS

- Pro
  - Flexible patterns of control
  - More efficient use of the CPU than in a traditional sequential system
  - Typically consumes less stack space than a traditional multi-tasking system
- Con
  - Responsiveness could be better - single event queue makes it difficult to prioritise work
  - Encapsulation difficult - event handlers must remember their state by using global variables

Active object computing model

- Active object computing model addresses the problems of TEDS
- Based on the “actor” model developed by Carl Hewitt and colleagues in the 70s and 80s
  - autonomous software objects communicating by message-passing
- UML specification includes the concept of active object
- Essential idea
  - Use multiple event-driven systems in a multi-tasking environment
  - Or as Samek puts it
    - Active object = thread of control + event queue + state machine

Active objects system



(Samek, 2008, p.267)

Characteristics of the active object computing model

- Asynchronous communication
  - Active objects receive their events exclusively via their event queues
  - Event producers post events to event queues but don't wait for response
  - No distinction between events generated by ISRs and events generated by active objects
- Run-to-completion
  - RTC semantics guaranteed by structure of each AO's event loop - `dispatch()` must complete and return before next event can be extracted from event queue
  - Notice that here RTC is *not incompatible* with a preemptive RTOS
    - RTC step can be preempted by another AO's thread of control (task) without any concurrency hazards, assuming that *AO's do not share resources*

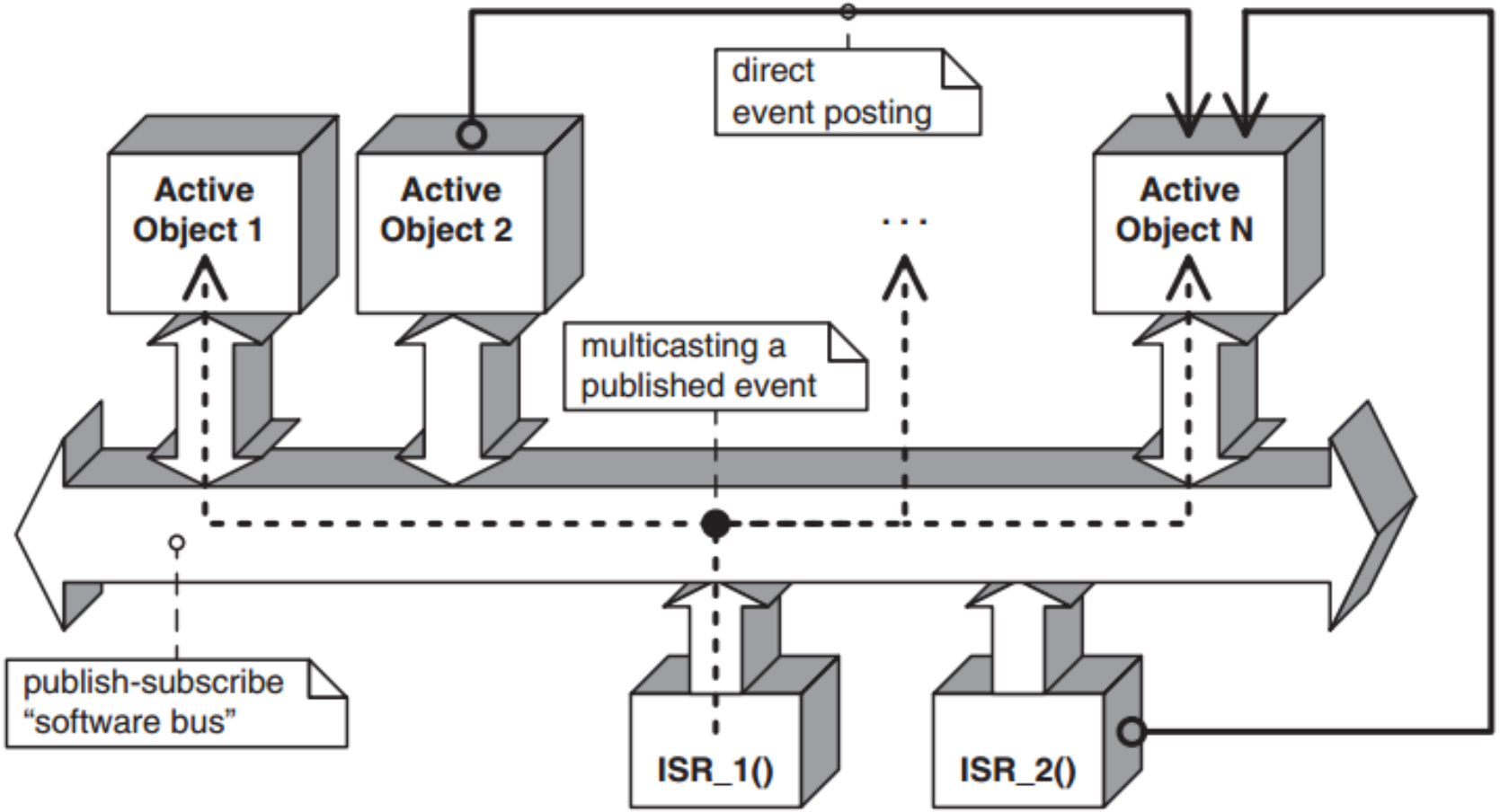
Characteristics of the active object computing model

- Encapsulation
  - All data and other resources are *local* to the active object, i.e. active objects have *no shared resources*
  - Communication with the outside world and with other active objects is restricted to asynchronous event exchange
  - Event exchange and the event queue are managed by the framework and are guaranteed to be free from concurrency hazards
  - Encapsulation does not require an object-oriented language - just a disciplined approach to *information hiding*
- Complements traditional multi-tasking RTOS
  - Most common implementation of active objects is to map them to tasks of a traditional preemptive RTOS, e.g. uCOS-II
  - But the active object model can also be implemented using a very simple cooperative (non-preemptive) scheduler

Event delivery mechanisms

- A key component of an event-driven framework is the *event delivery mechanism*
  - Responsible for the efficient transfer of events from producers to consumers
- Two common types of delivery:
  - *Direct event posting*: producer of the event puts it directly into the event queue of the consumer
  - *Publish-subscribe*: producer "publishes" events to the framework, the framework delivers the event to the event queues of all active objects that "subscribed" to it

Event delivery mechanisms



(Samek, 2008, p.280)

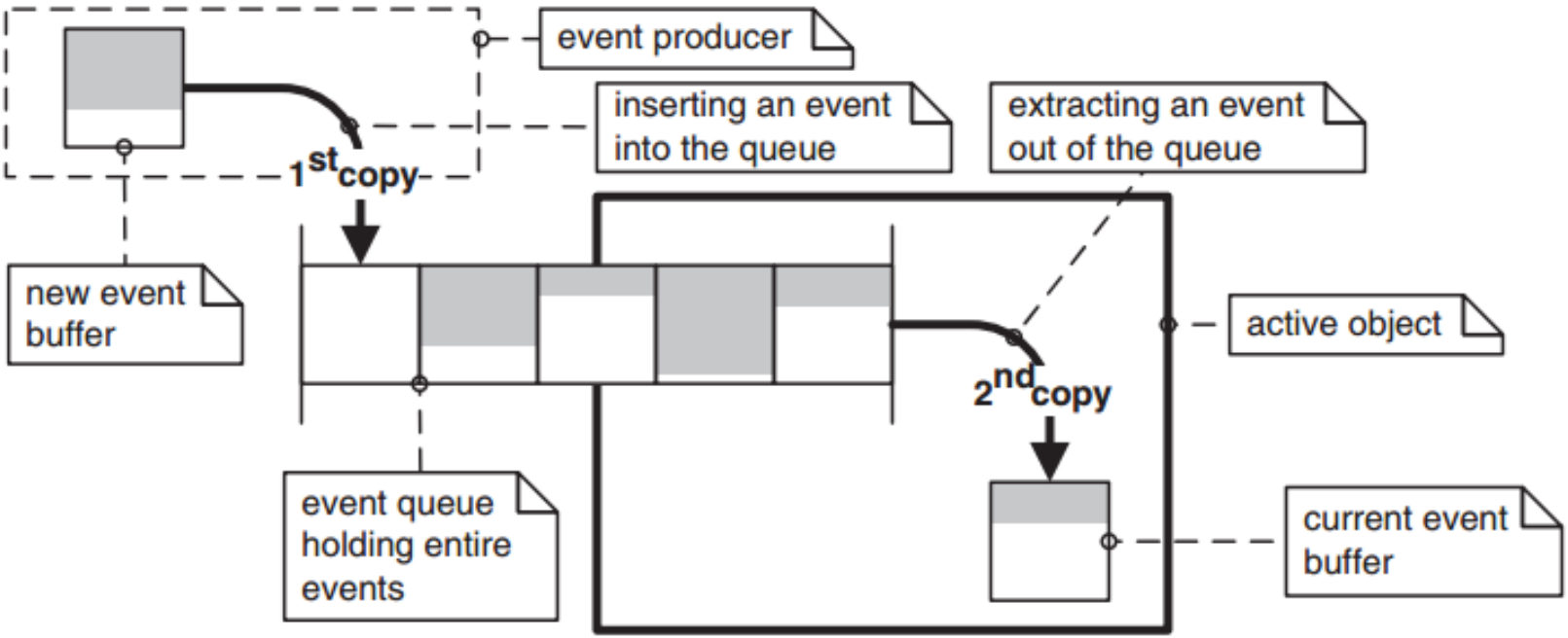
Event delivery mechanisms

- Direct event posting
  - Simple to implement
  - Push-style communication: active objects receive events whether they want them or not
  - Tight coupling between producers and consumers
- Publish-subscribe
  - Producers and consumers are loosely coupled
  - A “mediator” is required to accept published events and deliver them to subscribers
  - The mediator must be capable of storing information about subscribers, allowing dynamic subscription and cancellation
  - Efficient, thread-safe multicasting of events is required

Event memory management

- Events are produced and consumed *dynamically* (and frequently)
- Dynamic production and consumption of events is the key function of an event-driven system
- Efficient management of the memory used by events is crucial, since this memory must be reused as new events are produced and old events are consumed
- A challenge for the event-driven framework is to ensure that event memory is not reused until all active objects have finished their RTC processing of the event

Event memory management - copy

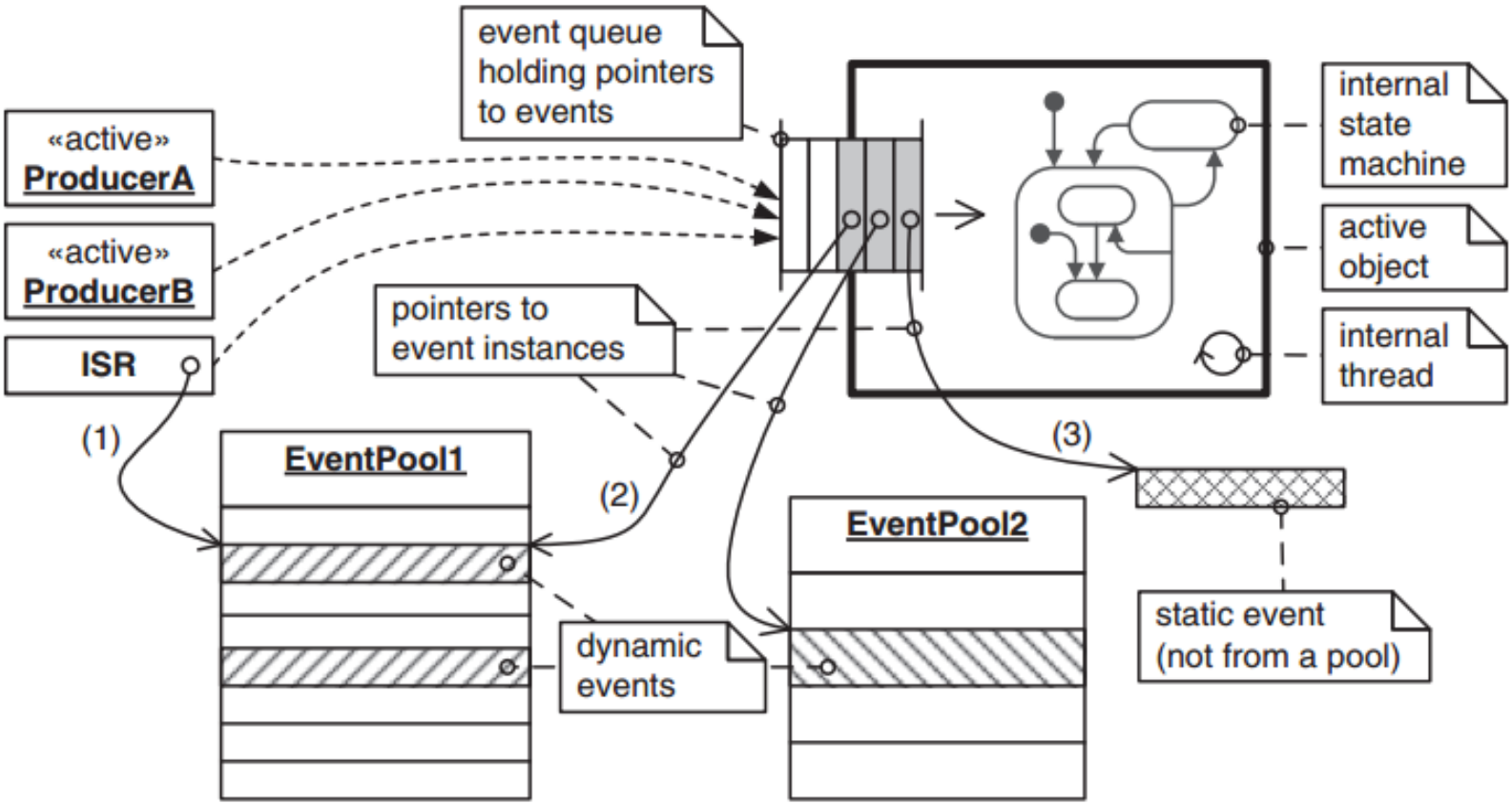


(Samek, 2008, p.283)

Event memory management

- This approach is *safe* - avoids corruption of event memory before the event has been processed
- But it's *expensive* - copying an entire event from the producer's event buffer into the consumer's event queue and then copying from the event queue into the consumer's event buffer introduces a significant overhead
- For this reason, it is common to store just a *pointer* to the event memory in the event queue
- The main problem with this approach is that the producer of the event needs to know when all consumers have finished with it before it reuses the event memory
- An event-driven framework can maintain control of the entire event lifecycle and so can use an efficient pointer-passing approach to event exchange and arrange for the safe *garbage collection* of event memory once all active objects have processed the event

Event memory management - zero-copy

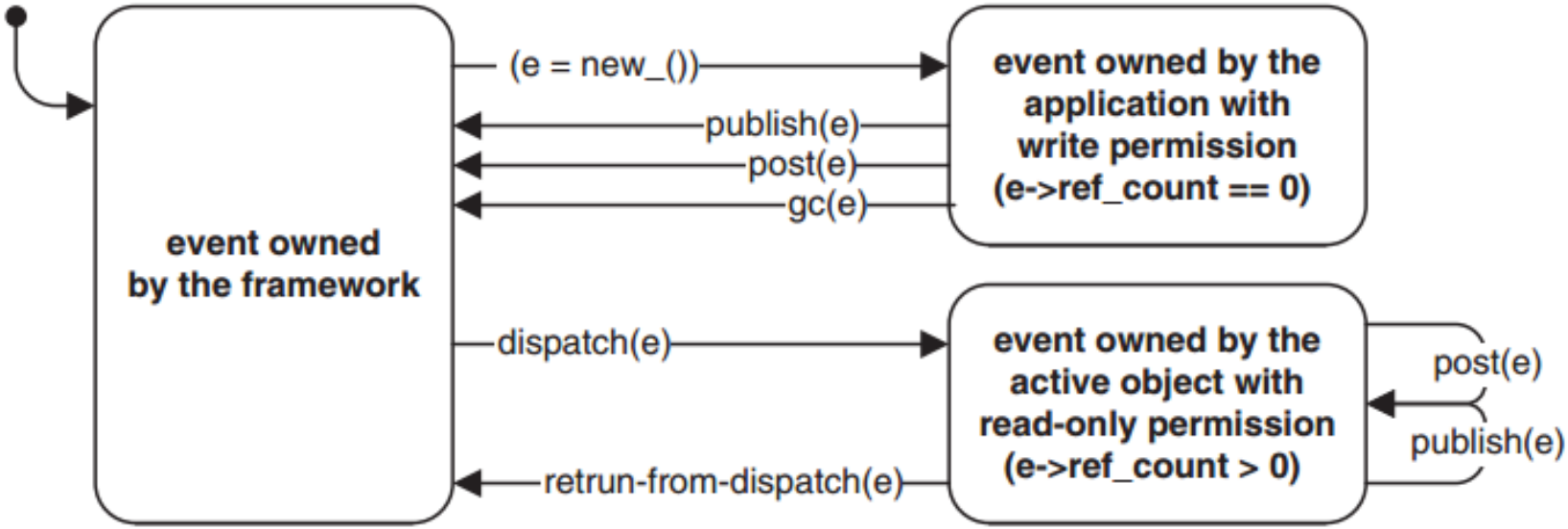


(Samek, 2008, p.285)

Event memory management

- uCOS-II provides exactly the right kind of features to make the management of event memory straighforward
- It offers *message queues* with thread-safe *pend* and *post* operations for pointer-sized data
- It offers *memory pools* for the allocation and deallocation of fixed size blocks of memory - avoids memory fragmentation (problem with malloc and free) ; offers deterministic performance
- Event-driven framework can implement a simple *reference counting* garbage collection algorithm to handle the multicasting of events in a publish-subscribe system
- However, the application code must follow a disciplined approach to *event ownership*

Event ownership



(Samek, 2008, p.288)

Acknowledgements

- Samek, M., Practical UML Statecharts in C/C++ Event-Driven Programming for Embedded Systems, 2nd Edition, Newnes, 2008