Thoughts on Event and Thread Mediated Control Architectures

Chris Gill
cdgill@cs.wustl.edu
www.cs.wustl.edu/~cdgill/OG_Feb01.ppt

Center for Distributed Object Computing
Washington University, St. Louis, MO

Objectives of This Talk

Describe Capabilities Achieved in Event Mediated Models

- *Static/Dynamic/Hybrid Scheduling and Dispatching*
- *Adaptive Admission Control and Scheduling Optimizations*

Highlight a Few Key Features of the RTSJ

- Threading and event handling models and evidence of their fundamental unity in the RTSJ under a more general perspective

Suggest a Few Milestones for Evaluating/Unifying These Models

- Define Behavioral Descriptors as a Carrier for Unification
- Identify Property Preserving Transformations
- Study Implementation Cost Implications (overhead, jitter, …)
- Study Programming Model Implications
  - Complexity, encapsulated (OBP/OOP) & cross-cutting issues (AOP), design patterns and pattern languages, property weavers

Adaptive Event Scheduling

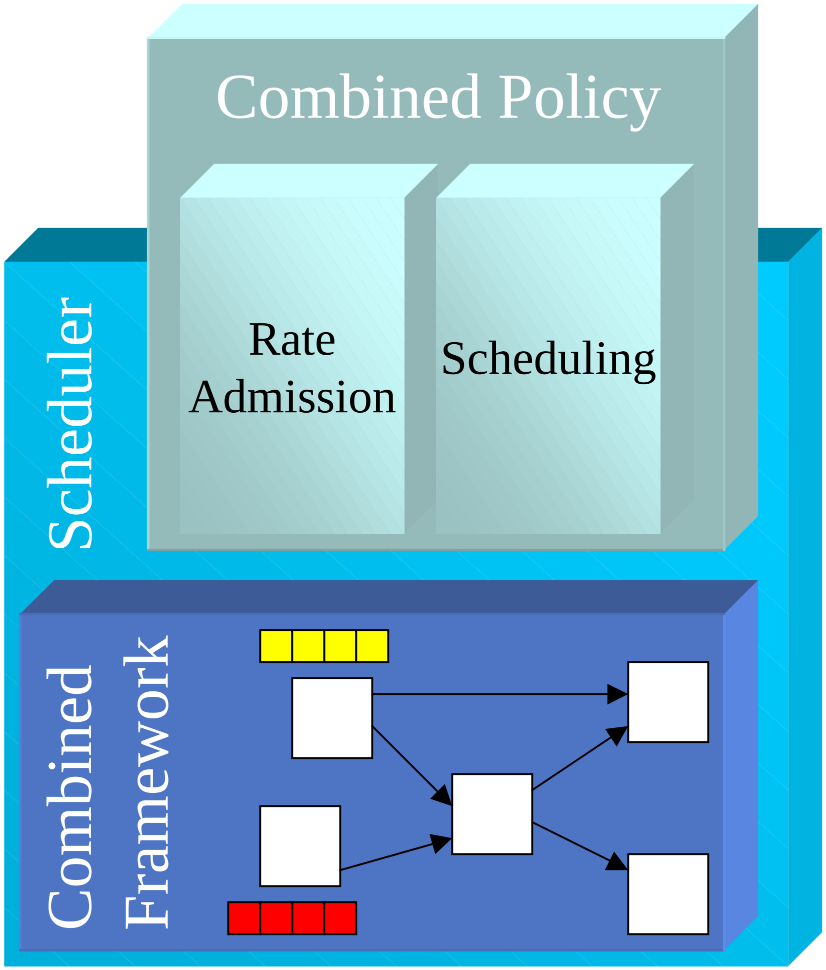
A little history

- AFRL/Boeing/HTC/WU ASTD program: measurements showed that strict layering of rate analysis / admission control *mechanisms* gave worst case bound no better than $O(n^2)$

Ideas

- Closer integration of *mechanisms* supports admission control during $O(n \log(n))$ or better sorting pass
- *Policy* layering is preserved: RTARM plugs a combined *policy* for schedule prioritization and admission control service requirements into the Scheduler's generic framework
- But, must enable/disable disjoint operations (and possibly operation dependencies) efficiently to reduce latency of adaptive transitions induced by mission state or RTARM

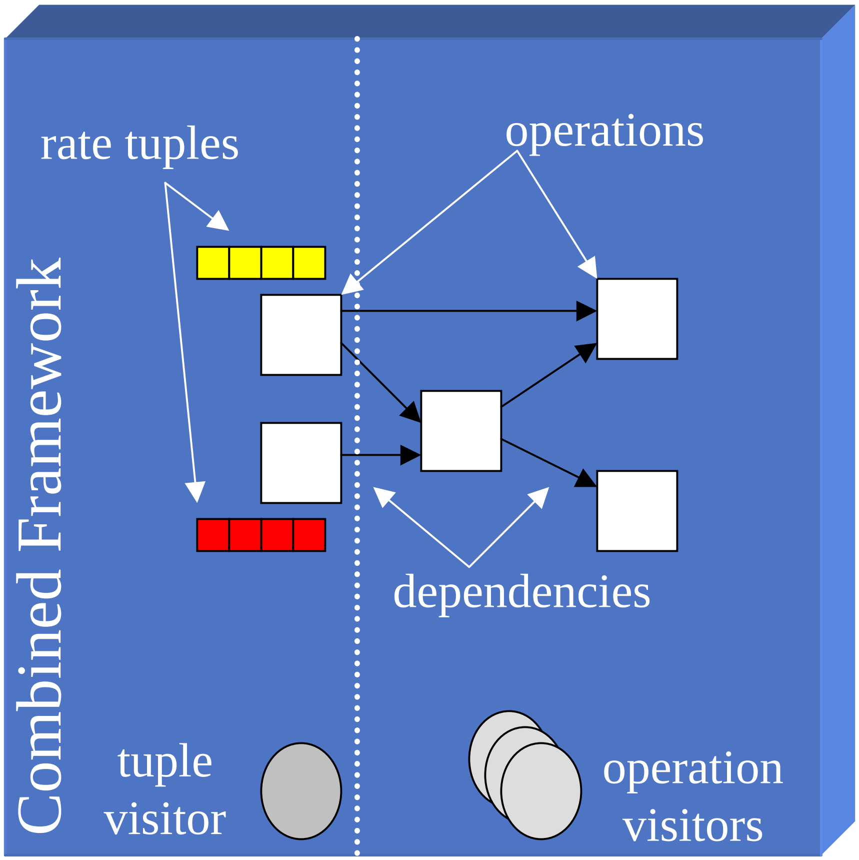
Scheduling/Admission Framework



New Framework Architecture

- **RT ARM plugs combined rate admission and schedule prioritization policy into scheduler**
- **Admission and schedule prioritization mechanisms in a combined scheduler framework enforce the policy requirements**

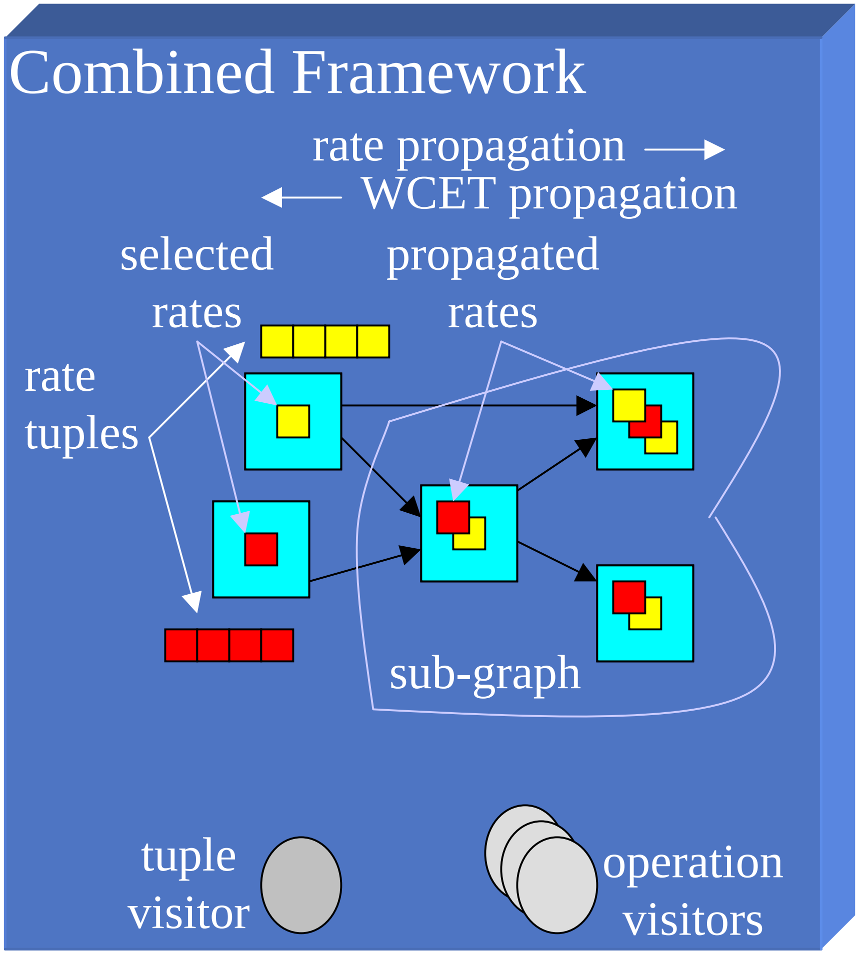
Framework Data Structures & Visitors



Framework Extensions

- **Rate tuples and visiting order index (sort-able pointer array) were added to data structures from dynamic TAO scheduler**
- **New dependency graph visitor was added to perform admission control over rate tuples**

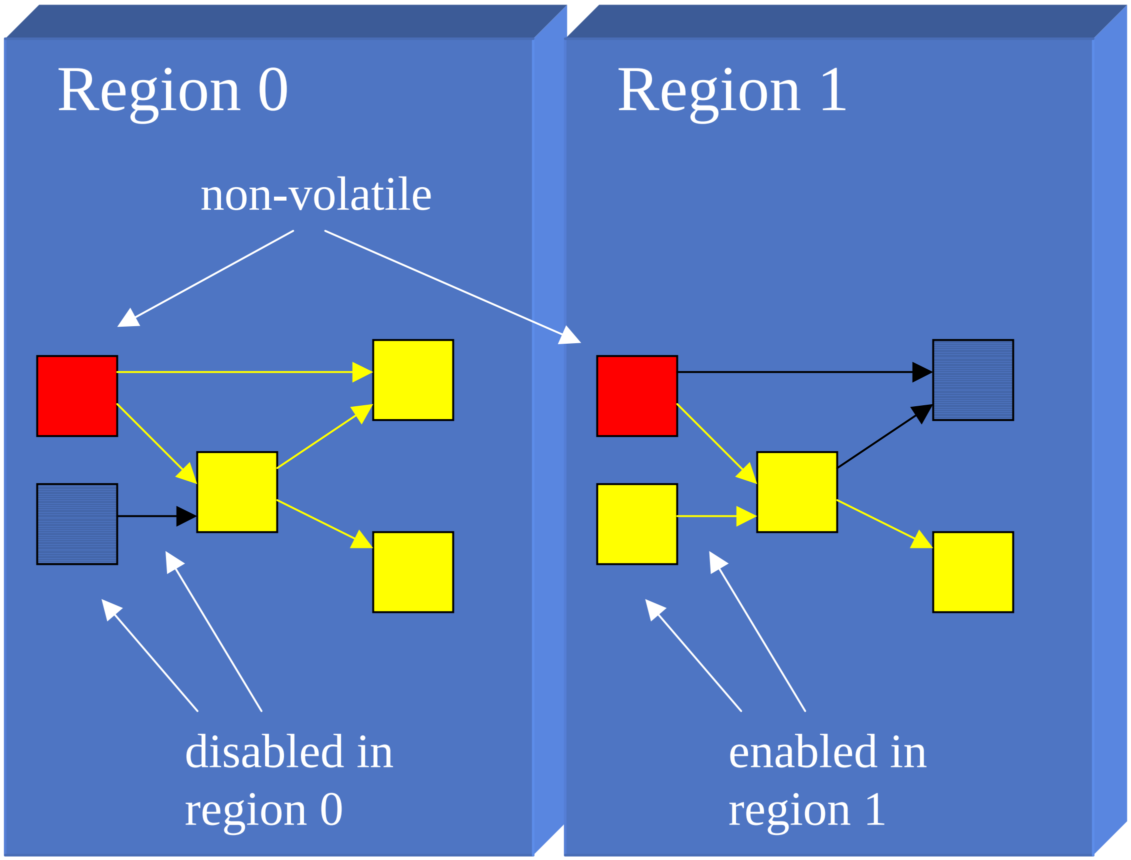
Schedule Computation Algorithm



Re-factored Algorithm

- **Reverse-propagation visitor sums WCET values up each sub-graph**
- **Tuple visitor chooses rates at "root" nodes**
- **Forward-propagation visitor does multi-set union of selected rates down each sub-graph**
- **Priority visitor assigns priorities to operations**

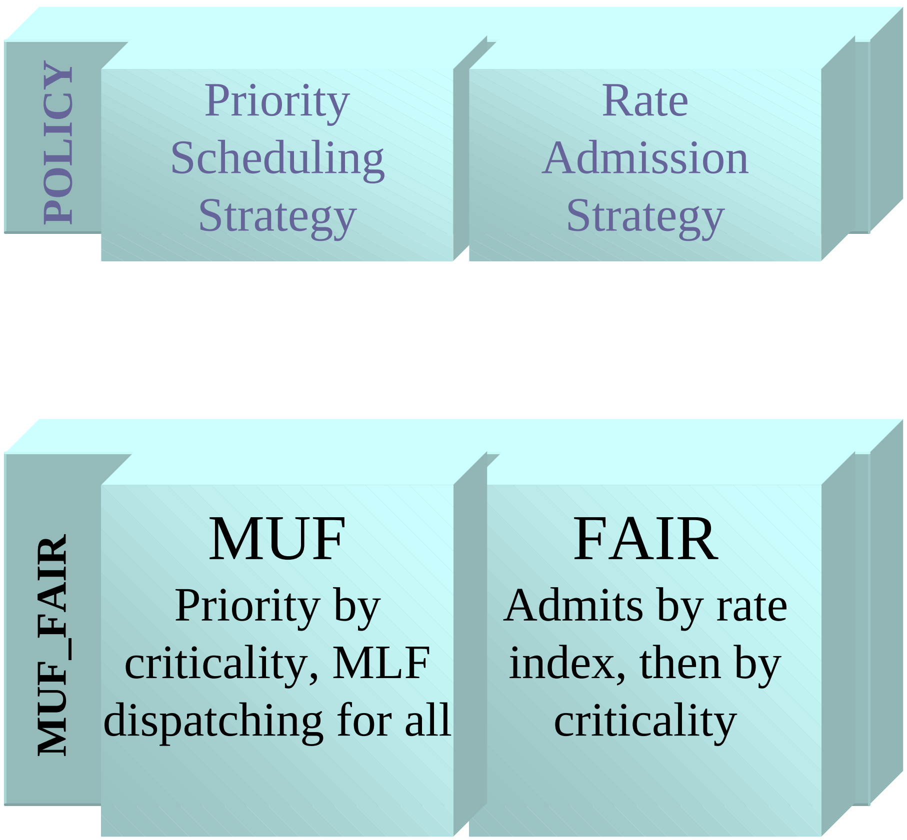
Disjoint Operations & Dependencies



Adaptive Transitions

- **Operation sets may differ between operating regions: add enable and disable behavior**
- **Internal EC operations must persist across regions: can mark as nonvolatile**
- **Automatically disable absent operations within the reset calls**

Scheduling/Admission Policies



Prototype Implemented

- **MUF_FAIR: Maximum Urgency First (MUF) scheduling policy + a new "Fair Admission by Indexed Rate" (FAIR) admission control policy**

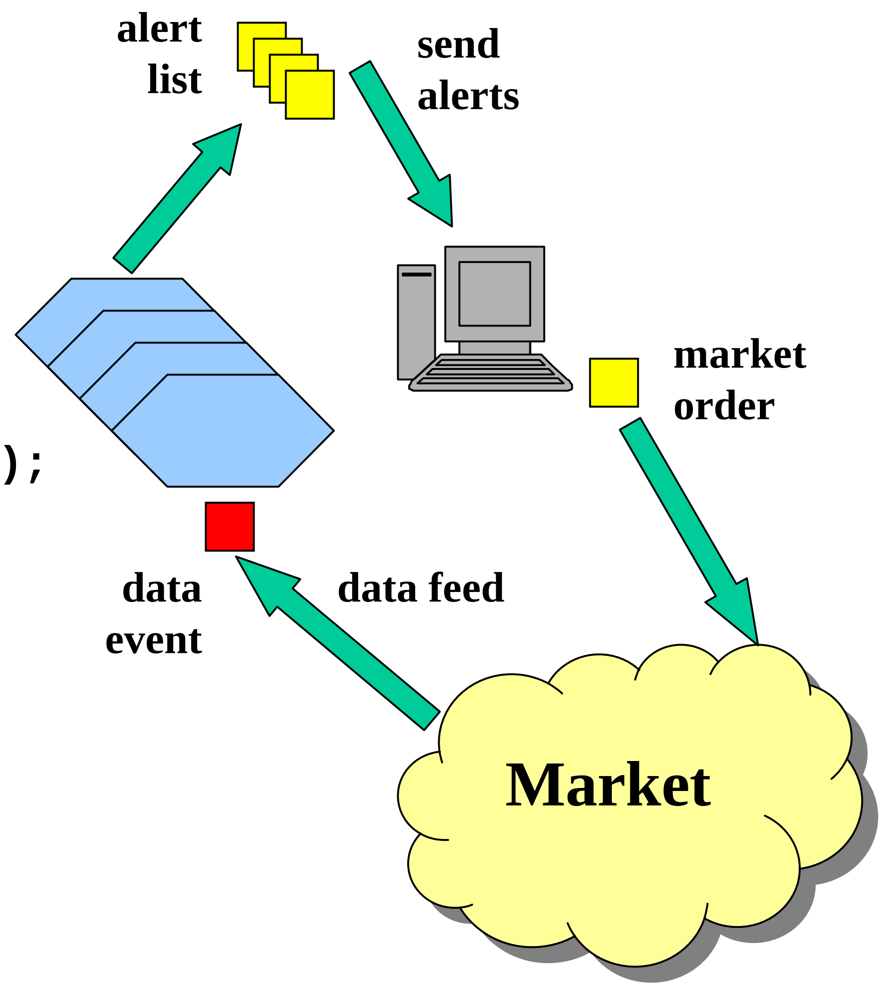
Key Observations

- **Release Characteristics parameterize static and dynamic execution eligibility and feasibility decisions (scheduling, dispatching)**
- **Other decisions (e.g., adaptive admission control) may modify release characteristics**
- **Complex interactions between decision points along the path**

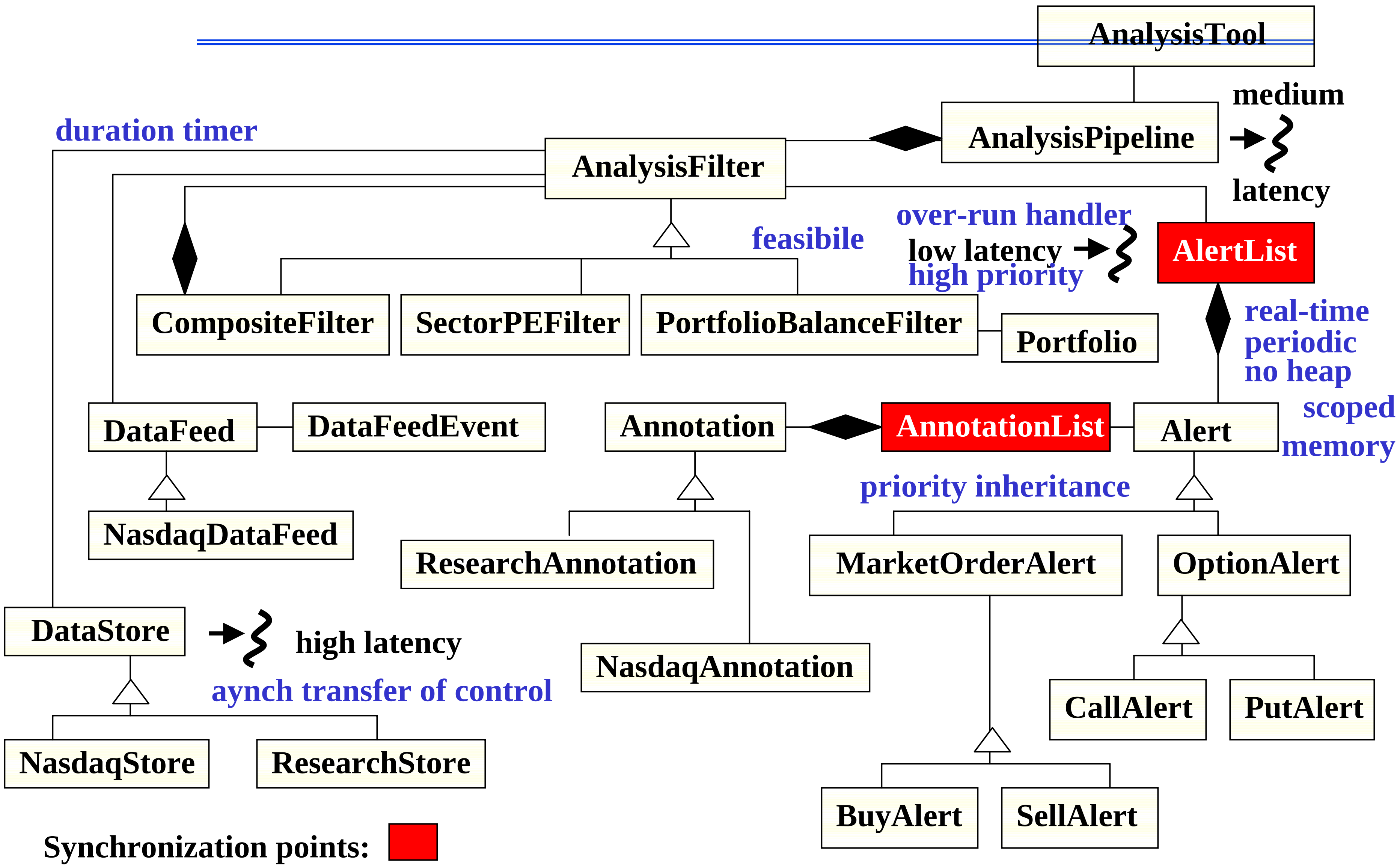
RTSJ Example: Stock Market Analysis Tool

```
public class AnalysisTool
{
  public static void main
    (String [] args)
  { AnalysisPipeline ap =
      new AnalysisPipeline ();
    ap.addFilter
      (new PortfolioBalanceFilter ());
    ap.addFilter
      (new SectorPEFilter ());

    ap.run ();  // run the pipeline
  }
}
```



RTSJ Example: Java/RTSJ Issues



Washington
WASHINGTON UNIVERSITY IN ST LOUIS

DOC

RTSJ: Release Characteristics Issues

```
public class AlertThreadAdapter implements javax.realtime.Schedulable

public AlertThreadAdapter ()
{ /* get/set release/memory/dispatch parameters ... */
  addToFeasibility ();}

 public void run ()
 {javax.realtime.RealtimeThread t =
    javax.realtime.RealtimeThread.currentThread ();
  for (;;)
  { t.waitForNextPeriod ();  // respect advertised cost, period
    pipeline.sendAlerts ();
  }
 }
}
```

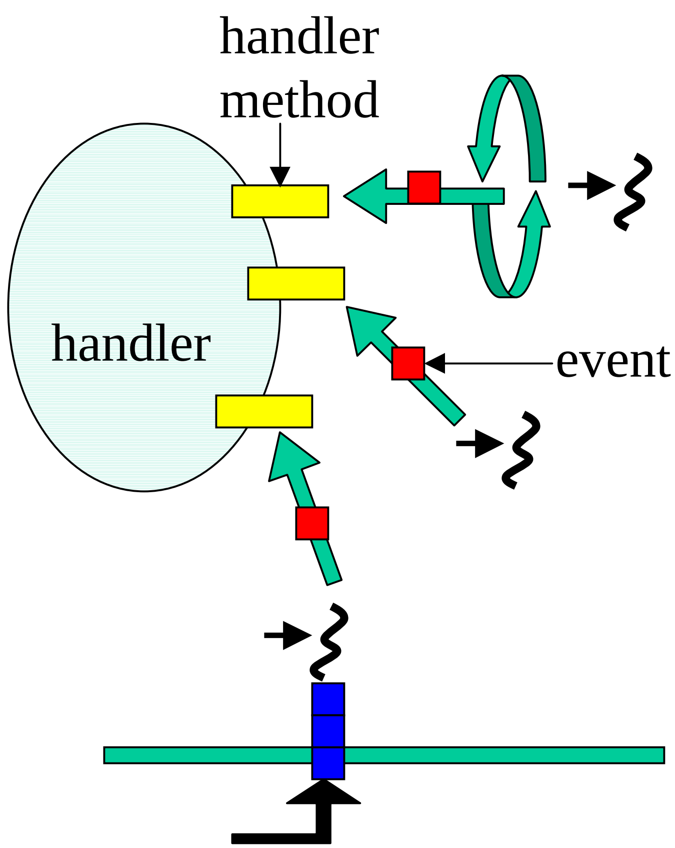
RTSJ: Time and Timer Issues

```
// A needed solution: watchdog timer
public class StoreTimeoutHandler
  extends javax.realtime.AsyncEventHandler
{public void handleAsyncEvent() {/* ... */}}

public class StoreThreadAdapter
  implements javax.realtime.Schedulable
{ public void run ()
  { // ... set up thread priorities ...
    long m = 60000; // one minute
    new javax.realtime.OneShotTimer
     (new javax.realtime.RelativeTime (m,0),
      new StoreTimeoutHandler ());
    store.annotateAlert (alert);
  } // ...
}
```

- **Threads offer a clean programming model**
- **However, many real-time systems benefit from asynchronous behavior**
- **Also, pacing is an effective/alternative way to reduce resource contention and improve resource utilization**

Event Handling Model



- **Threads allow synchronous programming styles**
- **Sometimes, asynchronous styles are more appropriate**
  - **Real-world timing issues**
  - **Decoupling processing**
- **Events-and-handlers model provides mechanisms for:**
  - **Synchronous actions (e.g., w/ threads)**
  - **Asynchronous actions (e.g., w/ timers)**
  - **Mixed (half-sync/half-async)**

RTSJ: Async Event Handling Issues

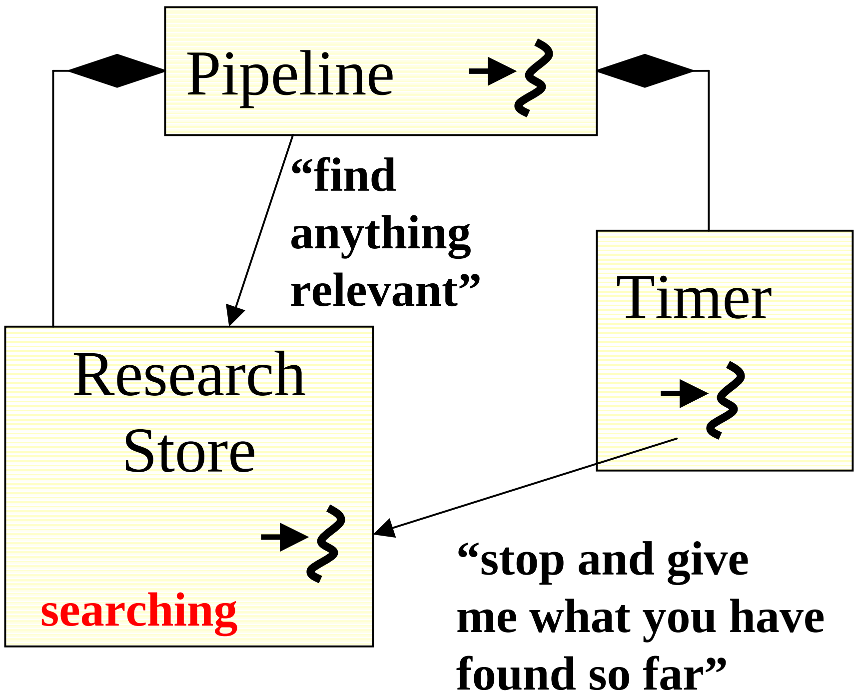
```
// Another way to implement periodicity

public class TransmitTimeoutHandler
  extends javax.realtime.AsyncEventHandler
{public void handleAsyncEvent () {/*...*/}}

new javax.realtime.PeriodicTimer
     (null,
      new javax.realtime.RelativeTime
            (1000, 0),
      new TransmitTimeoutHandler ());
```

- **Previous example of a one-shot timer used to determine when a long-running thread had been gone too long**
- **Could also use a periodic timer to re-implement the high priority alert transmission code**

RTSJ Issues: Async Transfer of Control



- **Want to provide real-time behavior for long-running synchronous activities (e.g., searches)**
- **For safety/fault-tolerance, some activities may need to be halted immediately**
- **However, standard threading and interrupt semantics can produce undefined/deadlock behavior in many common use-cases**
- **ATC refines semantics**

RTSJ Issues: Async Transfer of Control

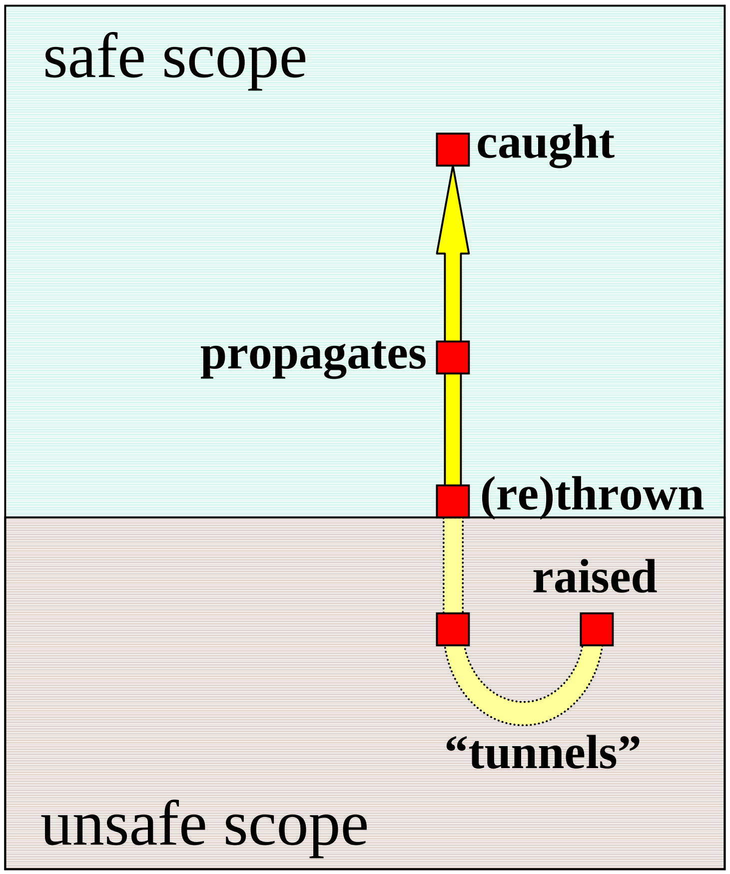
```
// Data Store Query Code
public abstract class DataStore
{ /* ... */
public abstract void
annotateAlert (Alert a)
throws javax.realtime.AsynchronouslyInterruptedException;
}
```

- **Even with the one-shot timer, the long running-thread must be reigned in somehow**
- **Deprecated Thread stop, suspend calls are unsafe**

```
// In timer handling for
// StoreThreadAdapter run ()
t.interrupt ();
```

- **ATC defers exception as pending in synchronized methods – avoids problem w/deprecated Thread stop method**

RT Issues: Exceptions



- **Additional special-purpose exceptions w/ standard semantics for**
  - **Memory management**
  - **Synchronization**
  - **System resource management**
- **Special semantics for ATC**
  - **When to throw (or not)**
  - **Deferred propagation semantics ("exception tunneling")**
  - **Nesting of scopes / exception replacement**

RTSJ: Exceptions Issues

- Semantics for AIE are different than others
  - deferred in pending state until inside a safe scope, where it will be thrown
- Other new exceptions deal primarily with incompatibilities of memory areas
  - Trying to assign a reference to scoped memory to a variable in immortal or heap memory
  - Setting up a WaitFreeQueue, exception propagation, etc. in an incompatible memory area
  - Raw memory allocation errors (offset, size)
  - Raw memory access errors
- What do we need to do with all this in a distributed context

Concluding Thoughts

Unifying the Models

- Straightforward to model a periodic remote invocation (or sequence of invocations) as a distributed thread
- DRTSJ release characteristics descriptor would need to describe locality (endsystem) as well as existing RTSJ attributes
- Similar generalizations seem useful (ATC for partial failures?)

More Difficult Questions

- One-to-many simultaneous invocation is often useful (scoped concurrency)
- Can describe as a DAG of thread spawns and joins
- But, how do we relate the thread-level descriptors, since same cascade repeats
- Also, where can/should we do synchronization (transactional safety?)

Programming Model Issues

- Middleware seems like an appropriate place to shield the engineer from complexity, while giving a substrate/receiver for weaving
- Goal: one completely unified model, or > 1 that are ***semantically*** unified?