REAL-TIME DISTRIBUTED SYSTEMS DETAILED DESIGN METHODOLOGY (3)

TASK STRUCTURING

The task structuring step is required for all real-time concurrent system design approaches (and it is common to the DARTS and COMET approaches). The task structuring criteria are based on heuristics determined through substantial experience with developing concurrent real-time systems.

Task Structuring Issues

The system can be behaviourally modelled as a set of concurrent active objects (or tasks, threads, processes, etc) and functions which communicate via discrete event flow and data flows (messages). The purpose of task structuring is to formalize the concurrent task boundaries and the communication/synchronization between them.

The primary basis for task structuring is the *asynchronous* nature of objects in the system, objects in the behavioural system model that can execute sequentially are grouped into one task, whereas objects that must execute concurrently are mapped to a single task.

Task Structuring Criteria

There are five main categories of task structuring criteria and any task should be structured using criteria 1, 3 and 4, or criteria 2, 3 and 4:

1. *I/O task structuring criteria* - examines how device I/O objects are mapped to I/O tasks and when an I/O task is activated.
2. *Internal task structuring criteria* - examines how internal objects and functions are mapped to internal tasks and when an internal task is activated.
3. *Task priority criteria* - examines priority assignment to tasks.

4. *Task clustering criteria* - examines whether and how objects should be grouped into tasks.
5. *Task inversion criteria* - used for merging tasks to reduce overhead.

There are two stages of application of these criteria - the first three are used to directly map objects in the analysis model to tasks in the design model, and the last two are used when refining the design model to meet various performance criteria. Once the tasks are defined, the task interfaces are specified.

I/O Task Structuring Criteria

Generally behavioural models do not consider hardware related characteristics regarding I/O devices, but this information is important to determine the characteristics of task interfaces to the devices.

I/O device characteristics:
- asynchronous - active or interrupt driven
- passive - read/write on a polled or demand basis
- communications link

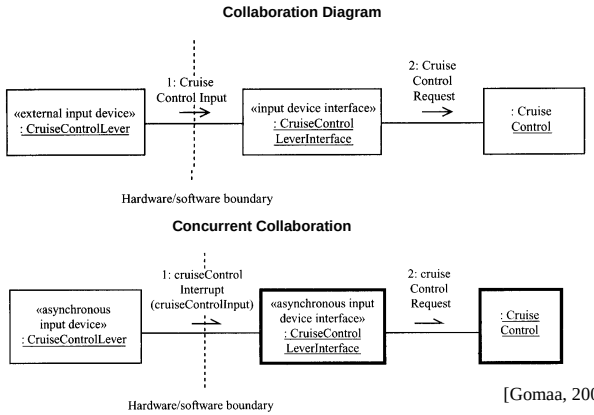
Data characteristics:
- discrete data - boolean or finite number of values
- continuous data - infinite values so usually polled

Passive device characteristics:
- sample the device on demand - sufficient to meet consumer needs?
- poll on a periodic basis - polling frequency?

Asynchronous I/O Device Interface Tasks

Each asynchronous I/O device has a task to interface to it (this simple concept can be traced back to Dijkstra, 1968 on CSP and Brinch Hansen, 1973 on OS design). An asynchronous device I/O task is usually implemented as an OS device driver task, which is activated by a low-level interrupt handler (or in some cases directly by the hardware).

Example - Cruise Control Lever object



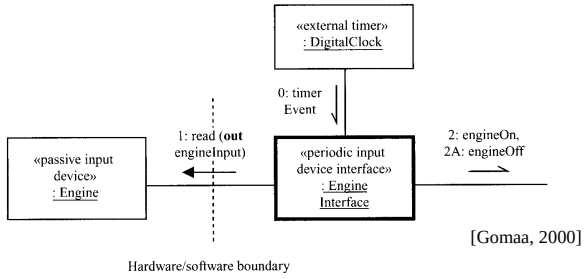
[Gomaa, 2000]

The cruise control lever is an asynchronous input device, so Cruise Control Lever is mapped to an asynchronous device input task. The task is activated by a Cruise Control Interrupt, then reads the input, converts to the internal format and sends it to the Cruise Control task as a message.

Periodic I/O Device Interface Tasks

These tasks deal with passive I/O devices that are used on a polled basis, i.e. they are timer activated. They are commonly used for acquiring data from passive sensors.

Example - Engine object



[Gomaa, 2000]

The current value of the engine sensor is read and an Engine Status Message created at every Timer Event.

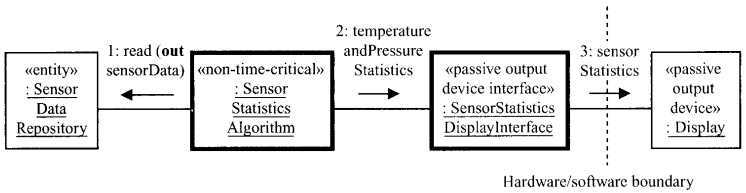
Timing considerations for periodic I/O tasks are influenced by the rate of change of the input data, e.g. for engine status sensing a 1 sec polling frequency may be adequate whereas for automotive brake sensing a 100 msec polling frequency may be required.

Note that it may be better to support an analogue input on a periodic polling basis rather than as an asynchronous device → as the rapid changing of values may generate an excessive load. For a digital device, a periodic input task is likely to consume more overhead (since the value of the sensor being monitored may not have changed since the last sample).

Passive I/O Interface Tasks

These tasks deal with passive I/O devices that are accessed on an a demand basis. They are commonly used for outputting data to passive output devices where there may be some benefit in overlapping output with the computational task that produced that output. They are less commonly used where overlapping of input data with the task that consumes that data is beneficial

Example - Sensor statistics display



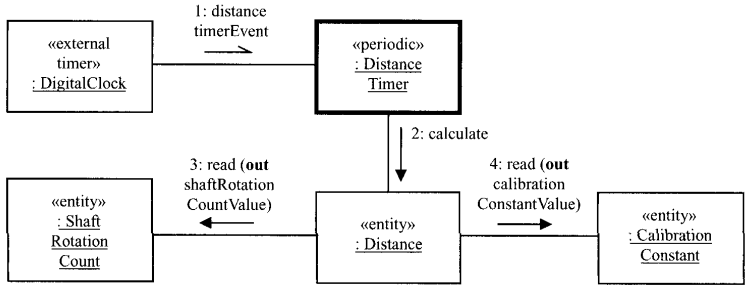
[Gomaa, 2000]

Resource Monitor Tasks

Devices that receive requests from multiple sources should have a resource monitor task to manage the requests even if the device is passive → controls the sequencing of requests and ensures data integrity, e.g. simultaneous output request to a printer.

Internal Task Structuring Criteria

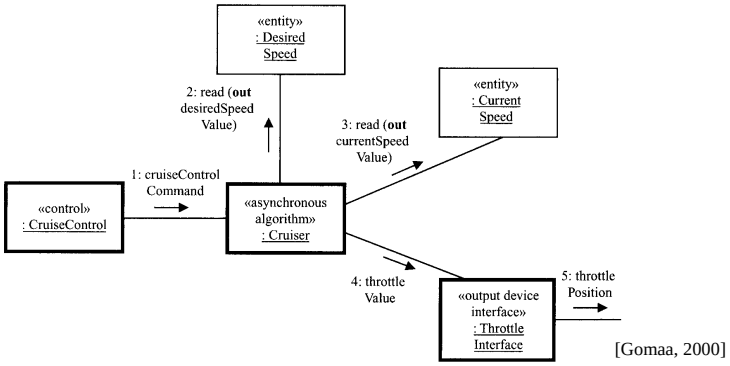
Periodic Tasks: every internal periodic activity is a candidate for structuring as a separate periodic task that is activated by a timer event, e.g:



[Gomaa, 2000]

The Distance Timer task is activated and requests the Distance object to read the Shaft Rotation Count and Calibration Constant to calculate the distance traveled periodically.

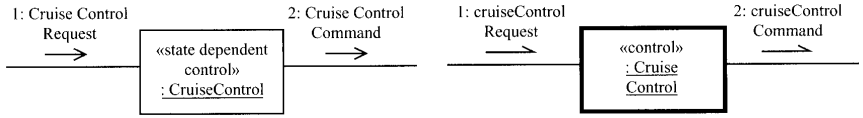
Asynchronous Tasks: an object that executes a function on demand is structured as a separate asynchronous task, i.e. it is activated by an external event or message, e.g. Cruiser object:



[Gomaa, 2000]

The Cruiser object is activated on demand by the arrival of Cruise Control Command messages. It reads the Desired Speed and Current Speed and then computes the required Throttle Value.

Control Tasks: because state-dependent control objects execute a statechart they are usually sequential so they can be executed by single control task, e.g. Cruise Control task:

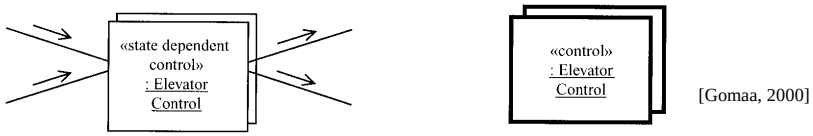


[Gomaa, 2000]

User Interface Tasks: because a user's interaction is a set of sequential operations, they can also be handled by a user interface or user role task. A user role task usually interfaces with the user through standard OS interfaces (since it is usually not necessary to develop special purpose I/O tasks to handle standard devices).

Different user interactions with the system may be supported by multiple user role tasks, e.g. an Operator Interface object in a factory application with multiple windows - examine factory status in one window and acknowledge alarms in another window.

Multiple tasks of the same type: there may be several objects of the same type → map to several tasks of the same type, e.g. a multiple elevator controller would have multiple Elevator Controller control tasks (which maintain identical statecharts with different state information).



[Gomaa, 2000]

Where there are too many objects of the same type to permit a direct mapping to multiple tasks, a task inversion strategy is followed.

Task Priority Criteria

Task priority assignment influences task structuring by identifying and separating out high-priority time-critical tasks. Generally control tasks (executing statecharts) and asynchronous I/O tasks are assigned a high priority. Non-time critical computationally-intensive tasks usually run at low priority. Assignment of priorities at this stage in the design methodology is preliminary and uses fairly coarse levels.

Task Clustering Criteria

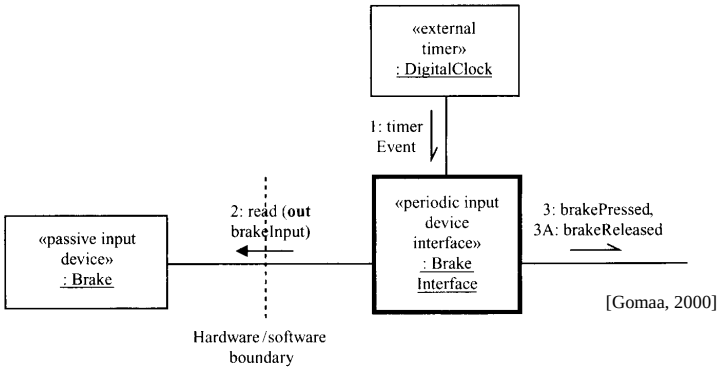
Concepts: the module cohesion or clustering concept has long been used in Structured Design approaches to identify the strength of bonding of the functions contained within a module. In DARTS, the task cohesion criteria were used to determine which objects and functions (i.e. the transforms in the behavioural model) could be grouped into a single task.

The behavioural model approach to hierarchical decomposition can lead to a large number of objects and functions which are potentially concurrent → can lead to a large number of tasks with associated complexity and execution overhead.

Task clustering or cohesion criteria provide a way of assessing the concurrent nature of the objects and how they should be grouped together e.g. two objects that are constrained so they cannot execute concurrently → no advantage in locating them in separate tasks.

Temporal Clustering: some objects may be activated by the same event even though there is no sequential dependency between them → objects can be grouped into the same task with an arbitrary execution order.

Example: monitor brake and engine sensors in the cruise control application.



[Gomaa, 2000]

If the sensors where asynchronous I/O devices → each would have a separate asynchronous I/O task. If the sensors are passive then two tasks, Monitor Brake and Monitor Engine, could be used that periodically poll (say every 100 msec) these sensors → combine into one task.

Issues in Temporal Clustering: there are tradeoffs to consider when structuring objects into a temporally clustered task:

1. Where one task is more time critical than another → map to separate tasks to allow different priorities.

2. If tasks are likely to be executed on different processors → should be assigned to separate tasks.

3. Preference in temporal clustering should be given to tasks that are functionally related and likely to be of equal importance from a scheduling viewpoint.

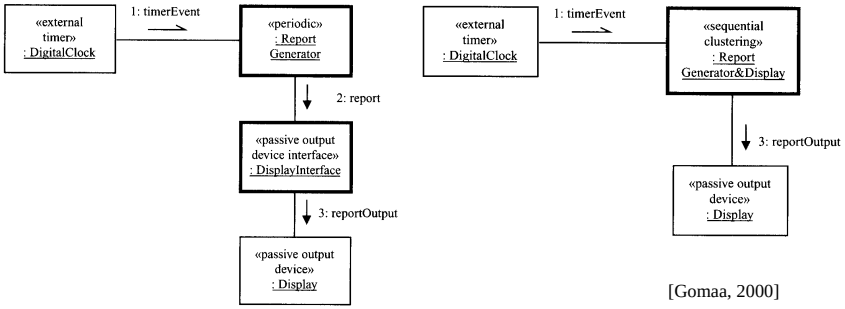
4. Even if two tasks have different periods, then can be combined into a temporally clustered task provided the periods are multiples of one another, e.g. one I/O task samples sensor A every 50 msec and another I/O task samples sensor B every 100 msec → the temporally clustered task has a period of 50 msec but samples sensor A every activation and sensor B on every second activation.

5. One group of tasks may be more important than another even if they have a common multiple in their periods → so should be left in a higher priority task, e.g: in the cruise control problem the two tasks Monitor Brake and Check Maintenance Need may have periods of 100 msec and 10 sec respectively → could be combined in a temporally clustered task but Monitor Brake is a far more important function.

An example where temporal clustering is deliberately taken to an extreme is the *Cyclic Executive* style of real-time programming [Glass, 1983], i.e. all periodically invoked objects are grouped into **one** task with different periodic events activating different objects. It has been shown that this approach usually increases the cost of the system and is very difficult to maintain in practice (but is not that uncommon in some implementations).

Sequential Clustering:

Objects that are sequentially linked may be combined in one sequentially clustered task, e.g:



[Gomaa, 2000]

The tasks Report Generator and DisplayInterface are sequentially clustered into a single task.

Issues:

- Where the last task in a sequence does not send an intertask message → terminates a potentially sequentially clustered group of tasks so could be sequentially clustered.

- If the next task in a sequence also receives input from another source → should be structured in a separate task.
- If the next task in the sequence could hold up the preceding task so that it could miss an input or a state change → should be structured as a separate lower priority task.
- If the next task in the sequence is of a lower priority and follows a time-critical task → should be structured as a separate task.

Control Clustering: this criterion can be used to group together objects in a single control task and objects that execute actions triggered by the statechart or activities enabled by the statechart:

1. State-dependent actions: an action implemented as an operation on another object that is triggered by a state transition and it starts and completes during the state transition → the object can also be included in the control task if all operations are executed within the thread of control of the control task.

2. State-dependent activities that are enabled (disabled) by the control object due to a state transition and disabled (enabled) by another state transition → because the control object and activity execute concurrently it should be structured as a separate task.

3. State-dependent activities that are triggered by the control object due to a state transition and execute for the duration of that state - several cases are possible:

   a) If the activity is deactivated by the control object so both could be active concurrently → the activity should be structured in a separate task.

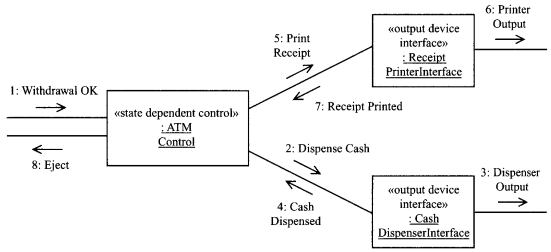
   b) Where the activity is the one that recognizes that it is time for a state change it will send an event to the control object, and if it is the *only* event that causes a state change → the activity could be clustered with the control object.

c) Where the control object could be activated by a number of events then the activities causing them should be structured as a separate task.

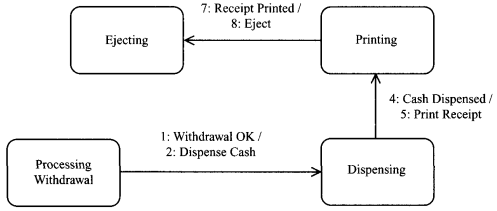
4. If there are messages from source objects sent to the control object as events that cause it to change state, then the source object may be clustered with the control object following the sequential clustering criteria.

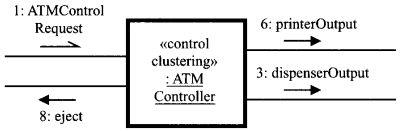
Example: ATM control

The ATM Control object executes the statechart and two actions are the triggering of *Dispense Cash* & *Print Receipt*.



From the statechart, both actions start and complete execution during a state transition → group both the output interface objects into the ATM controller task via the control clustering criteria.

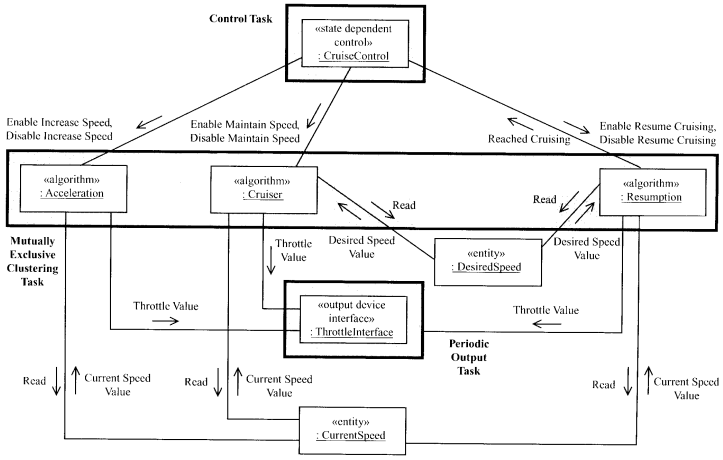




[Gomaa, 2000]

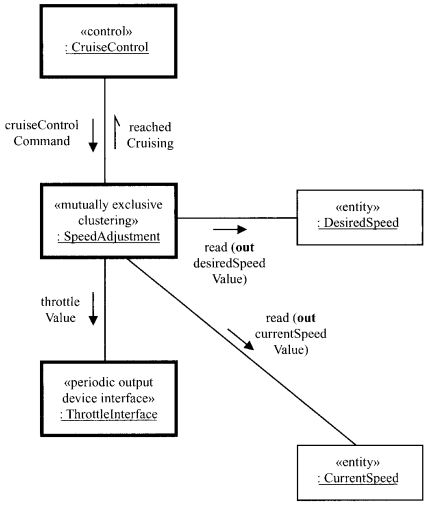
Mutually Exclusive Clustering: this criteria applies where a number of tasks are related but can only be executed one at a time so they can be grouped into a single task.

Example: Cruise Control



[Gomaa, 2000]

The three algorithm objects, *Acceleration*, *Cruiser* and *Resumption* are enabled on entry to a state and disabled on exit, so they are candidates for separate tasks. However these activities are all mutually exclusive and there is no advantage is structuring these objects as separate tasks:



[Gomaa, 2000]

Task Architecture Development

Guidelines for applying the task structuring criteria in the following sequence:

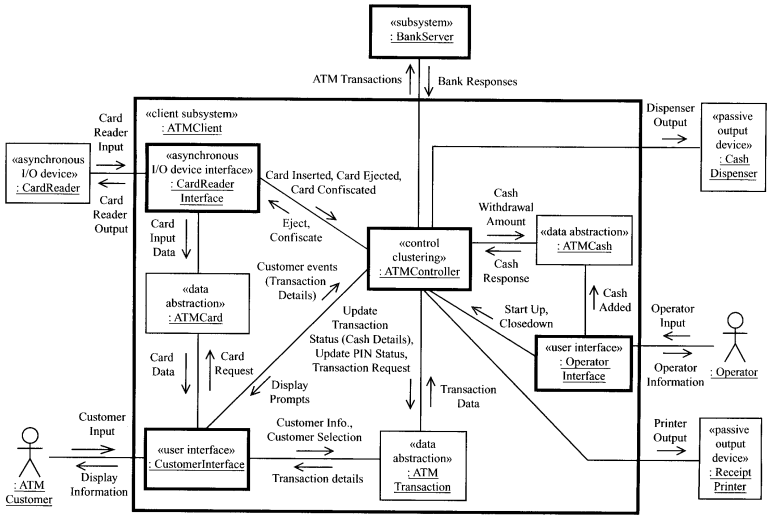
1. *Device interface tasks* - structure the device interface objects that interact with the external world into: asynchronous I/O, periodic I/O, passive I/O, resource monitor or temporally clustered periodic I/O tasks.
2. *Control tasks* - each state-dependent control object and all interacting objects that interface to it, are analyzed. Any object executing an action triggered by the control task can often be structured into the control task, whereas any activity enabled and later disabled by the control task should be structured in a separate task. If there are multiple control objects of the same type then their possible integration in one task should be examined.
3. *Periodic tasks* - analyze the internal periodic activities and if any are triggered by the same event or are sequentially executed → can be grouped in the same task.
4. *Other internal tasks* - look for any internal tasks that can be grouped into a task by the temporal, sequential or mutually exclusive clustering criteria.

In some cases multiple criteria need to be applied, e.g. consider a device interface object that is assigned to a device interface task initially, but:

- suppose that it is activated by a control object, and it is synchronously performed during a state transition → re-examine the initial task structuring decision as the device interface object could now be combined in control task via control clustering criteria.
- suppose the device interface object interacts with a high-priority task as well → re-examine again as it should be a separate task via temporal clustering criteria.

The Initial Concurrent Collaboration Diagram

Based on the task structuring criteria just examined, an initial concurrent collaboration diagram (or Task Architecture Diagram) can be produced which identifies all tasks in the system, but has yet to define detailed task interfaces, e.g:



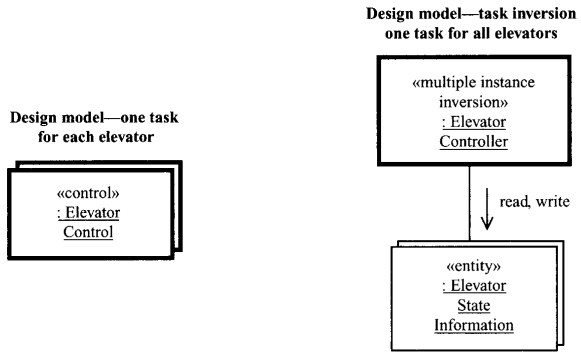
[Gomaa, 2000]

Design Restructuring using Task Inversion

The task inversion approach was first used in the Jackson Structured Development (JSD) method, and is a set of guidelines for reducing the number of tasks in a systematic manner. It is a design restructuring approach which is applied where there is some concern about the system overhead caused by a large number of tasks.

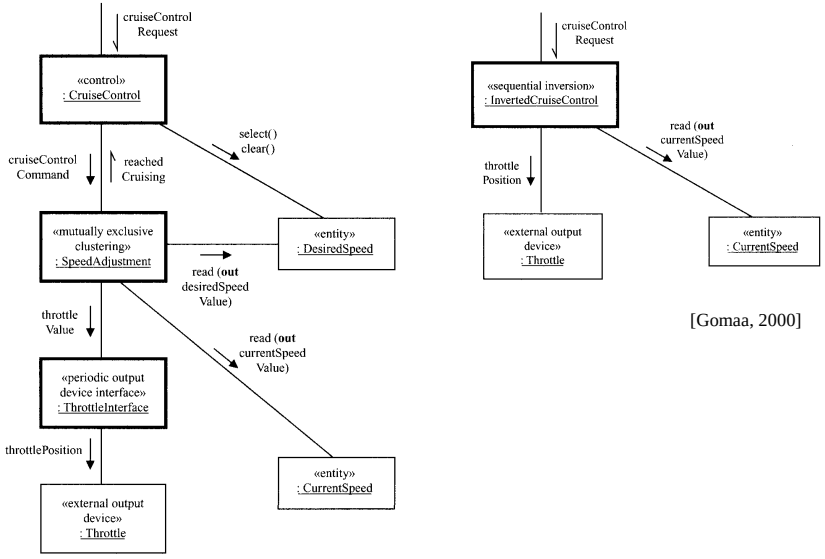
Multiple Instance Task Inversion: the conventional task structuring guideline for handling several objects of the same type is to create separate tasks of the same type → problem is that this may lead to unacceptably high overhead.

Task inversion replaces all these tasks by one task, but maintains a separate data structure for each object, e.g. the Elevator Controller handles multiple elevators:



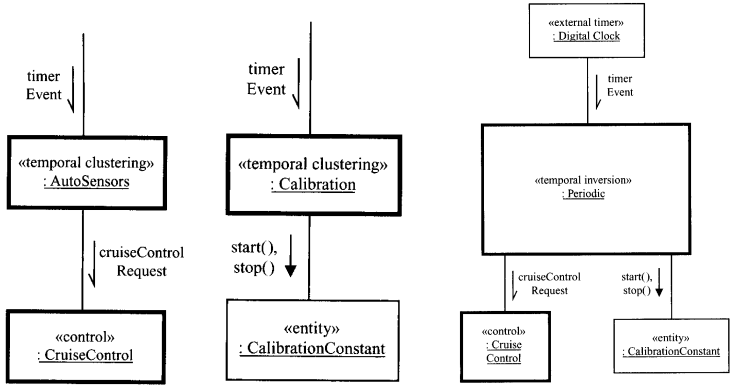
[Gomaa, 2000]

Sequential Task Inversion: used mostly where there is tightly-coupled communication between a producer task and consumer task → can combine tasks or "invert" the producer task with respect to the consumer task, e.g. the Cruise Control problem:



[Gomaa, 2000]

Temporal Task Inversion: usually used where two or more periodic tasks are combined into one task, so that the task has a scheduling procedure activated by a single timer event. Grouping objects that are not functionally related is not desirable from a design viewpoint, but is justified when optimization is necessary, e.g. the Cruise Control problem:



[Gomaa, 2000]

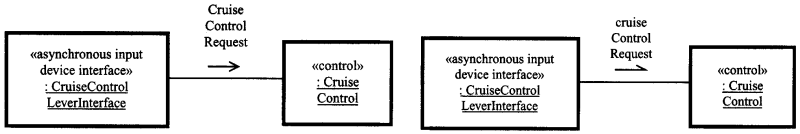
Two temporally clustered periodic I/O tasks, *Auto Sensors* and *Calibration* are combined into one temporally inverted task which is activated by the same *timerEvent*. The task checks which procedure is to be run.

Task Communication and Synchronization

The interfaces between tasks are simple messages on the initial analysis model collaboration diagram. These messages must be mapped to *task interfaces* in the form of message communication, event synchronization, or communication via information hiding objects.

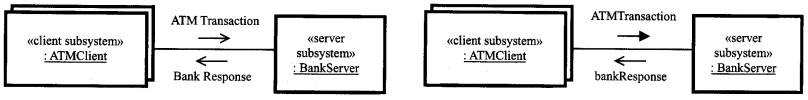
Message Communication:

- Loosely-coupled (asynchronous) - consumer employs a message queue and no response is required from the consumer, e.g. in the Cruise Control system the *CruiseControlLeverInterface* sends a message to the *CruiseControl* task without waiting. The *CruiseControl* task is then able to respond to messages from other sources.



[Gomaa, 2000]

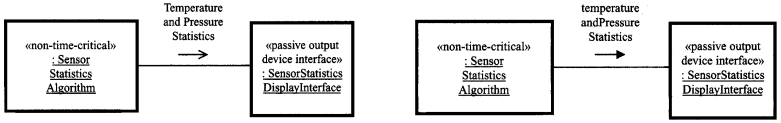
- Tightly-coupled (synchronous) with reply - the producer task waits for reply from the consumer task with no message queue, e.g. the ATM system:



[Gomaa, 2000]

The *ATMClient* task is unable to continue until the ATM Transaction is processed by the *BankServer*.

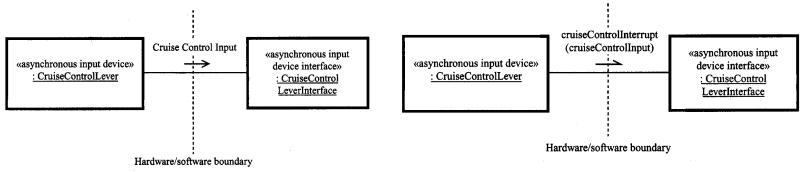
- Tightly-coupled (synchronous) without reply - the producer does not wait for a reply from the consumer task, e.g. a sensor statistics processor and display task:



[Gomaa, 2000]

Note that messages are not queued because there is no reason to compute sensor statistics if they cannot be displayed. Also as the *SensorStatisticsAlgorithm* task is only suspended until the *SensorStatisticsDisplayInterface* accepts the message → potential to overlap execution of the former task (compute bound) and the later task (I/O bound).

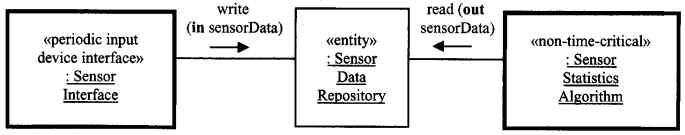
- Event Synchronization - there are three types: external event synchronization, internal event synchronization, and timer event synchronization, e.g. an external event (typically a hardware interrupt):



[Gomaa, 2000]

Timer events and internal events are similarly mapped to asynchronous messages without inputs.

Task Communication via Information Hiding Objects: tasks can exchange information via accessing shared information hiding objects, e.g. the *SensorStatisticsAlgorithm* task accesses *sensorData* placed by the *SnesorInterface* task via the *SensorDataRepository*:

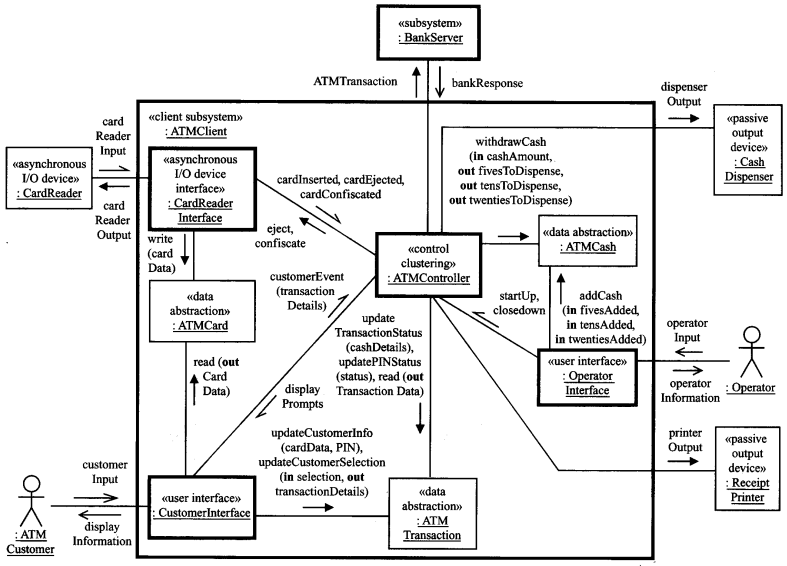


[Gomaa, 2000]

Note that although this is shown notationally as a synchronous message, accessing passive objects is implemented as operations on the object with appropriate send and return parameters.

Revised Task Architecture

The concurrent collaboration diagram can now be revised to show the complete task architecture: all concurrent objects and the interfaces between them e.g. the ATM bank system:



[Gomaa, 2000]

Task Behaviour Specification

The Task Behaviour Specification (TBS) is usually written in a structured form and contains the following information:

a) *Task interface* - includes a definition of message inputs and outputs, i.e. type of interface, name and parameters; definition of events, i.e. type of event, event name; external inputs and outputs.

b) *Task structure* - the task structuring criterion used to design the task and the mapping from the analysis model.

c) *Timing characteristics* - frequency of activation (period T_i), estimated execution time for task (C_i) and all paths through it.

d) *Relative priority* - priority in relation to other tasks.

e) *Event sequencing logic* - how the task responds to messages or events, i.e. what outputs are generated $\rightarrow$ could be in the form of a statechart.

f) *Error handling* - possible errors detected.

Example - TBS from ATM Bank System

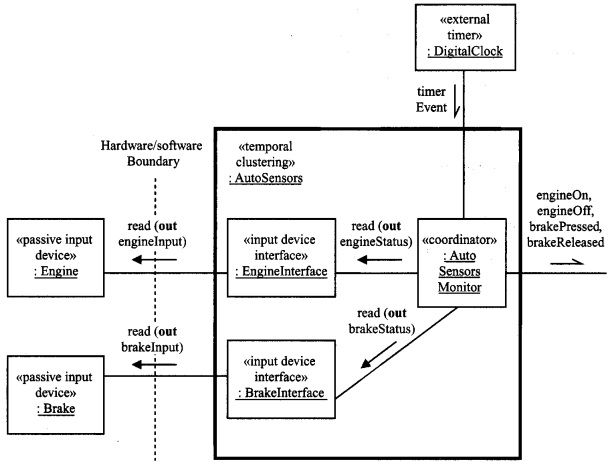
```
TASK:  Bank Server
a) TASK INTERFACE:
     TASK INPUTS:
       Tightly Coupled Message Communication with Reply
       Messages:
       1) validatePIN
          Input Parameters: cardID, PIN
          Reply: PINValidationResponse
       2) withdraw
          Input Parameters: cardID, account#, amount
          Reply: withdrawalResponse
       3) query
          Input Parameters: cardID, account#
          Reply: queryResponse
       4) transfer
          Input Parameters: cardID, fromAccount#, toAccount#,
amount
          Reply: transferResponse
     TASK OUTPUTS:
     Message replies as above

b) TASK STRUCTURE:
          Criterion:  Sequential Clustering
     Objects mapped to task:
     BankTransactionServer, PINValidationTransactionManager,
     WithdrawalTransactionManager, QueryTransactionManager,
     TransferTransactionManager,CheckingAccount,SavingsAccount,
     DebitCardAccount, TransactionLog

c) TIMING CHARACTERISTICS:
     Activation: Asynchronous – message arrival from clients
     Worst case inter-arrival time = 100 msec
     Average inter-arrival time > 1 second
     Execution time Ci: 10 msec per message

d) PRIORITY:
     High – needs to be responsive to incoming messages

e) TASK EVENT SEQUENCING:
     See statechart or pseudocode description

f) ERRORS DETECTED:
     Any unrecognized messages
```

Detailed Task Design Issues

Tasks typically contain a mix of internal object types. The task (which is an active object) is activated by an external or internal event, or a timer event. It then accesses the internal passive objects (that have been instantiated from a class). There are two possibilities:

- Object is only accessed by one task → it can be nested in that task
- Object is accessed by more than one task → it must reside outside the task (and the object must provide its own synchronization capabilities).

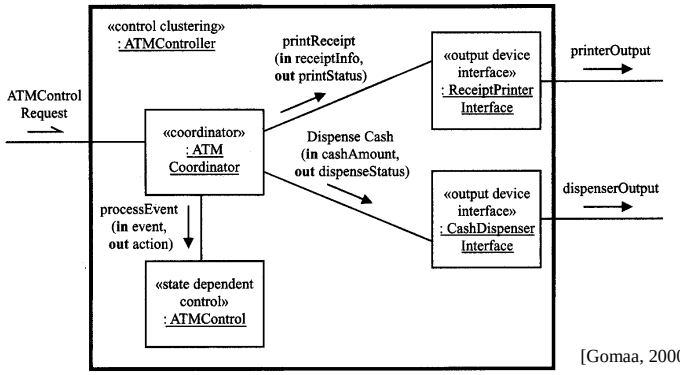
A **composite task** encapsulates a number of nested objects, and can be shown on the concurrent collaboration diagram. The same clustering criteria that are used to decide on object clustering in tasks are used to decide on the extent of object nesting in tasks, e.g. device interface objects included in the only task that accesses them:



[Gomaa, 2000]

The *AutoSensors* task temporally clusters the coordinating object *AutoSensorsMonitor* which accesses the device interface objects and produces the state information required by the *AutoControl* task.

Another example is the clustering of a control object (a state-dependent object) with output device interfaces:



[Gomaa, 2000]

The *ATMControl* object executes a statechart in response to the *ATMCoordinator* object which outputs to the output device interface objects.

The aim in creating composite tasks is to separate concerns of **how** a device is accessed in the device interface objects from **when** the device is accessed in the task. Thus the device interface can be reused in different applications by different types of task, and the task can use different types of devices by changing the device interface.

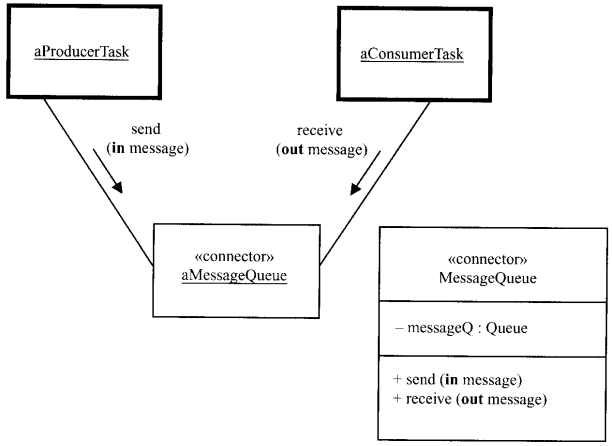
Access Synchronisation

When objects are located outside tasks and can be accessed by multiple tasks, then synchronization must be built into the object (hence the class it is instantiated from). A number of mechanisms can be used to implement mutual exclusion, e.g. semaphores, mutexs, monitors. Some capabilities are providing by the operating system, and others by the implementation language. At the detailed design level, a pseudo-code description of the synchronized data access operations is adequate.

Inter-task Communications

Detailed design of inter-task communications can be done through the specification of **connector** classes to encapsulate the implementation mechanism of loosely and tightly-coupled message communications.

For example, neither Ada or Java provides loosely-coupled message communications, so a *Message Queue* class can be used to provide a dedicated connector class, e.g:



[Gomaa, 2000]

Other connector classes can also be defined to include buffers and responses for tightly-coupled inter-task communications. Cooperating tasks can then use the required instantiated connector objects for inter-task communications.

Task Event Sequencing Logic

The final step in detailed task design is to define the behaviour of the object in the task that is responsible for all message handling and event sequencing (e.g. the coordinating object). These may defined in pseudocode or a statechart diagram.

e.g. Event Sequencing Logic for the *aProducer* task using the connector class defined earlier:

```
loop
     Structure message with name and optional parameters
     aConnector.send(message);
endloop;
```

e.g. Event Sequencing Logic for the *aConsumer* task using the connector class defined earlier:

```
loop
     aConnector.receive(message);
     Extract message with name and optional parameters
     case message of
          messageType1:
               objectA.operationX (optional parameters);
          . . .

          messageType2:
               objectB.operationY (optional parameters);
          . . .
      endcase;
endloop;
```

Summary of COMET design methodology steps

1. Software Architecture design
   - System and subsystem decomposition

2. Task Architecture design
   - Clustering criteria

3. Object and class design (information hiding passive objects)
   - Separation of concerns

4. Detailed task design
   - Nesting passive objects into composite tasks
   - Access synchronization of passive objects
   - Detailed inter-task communications
   - Event sequencing logic