REAL-TIME DISTRIBUTED SYSTEMS DESIGN METHODOLOGIES

SOFTWARE LIFE-CYCLE ISSUES FOR REAL-TIME SYSTEMS

Common software development models:

- Waterfall model - has some limitations but most-widely used
- Throwaway prototyping model
- Incremental development model (evolutionary prototyping)
- Spiral model

Waterfall model major stages (variations for RT systems):

- Software Requirements Specification (SRS) - specify the systems external behaviour. Since RT systems are usually a component of a much larger system → a System Requirements Specification usually proceeds the SRS.
- Software Architectural Design - for RT systems the separation of functionality into concurrent tasks is a major activity in this phase. This is in addition to the usual separation into function software modules. Behavioural and dynamic aspects of system performance are also considered in this phase.
- Detailed Design - algorithmic details of each system component is defined using a PDL (structured English or pseudo-code). For RT systems attention is paid to algorithms for resource sharing, deadlock avoidance, and interfacing to I/O devices.
- Coding - usually a concurrent programming language is selected (e.g. Ada, Modula2 or Occam) or a sequential language with a suitable multi-tasking operating system.

- Testing - Although the basic testing approach is similar to conventional systems testing, **the non-deterministic nature of RT systems introduces another level of complexity**, as does their application in embedded systems → may require environment simulators to be constructed.

  Testing can be subdivided into:
  - unit testing
  - integration testing
  - system testing
  - acceptance testing

  The lower and upper levels of the testing program have much in common with conventional software system, but integration testing must specifically test the concurrent task interfaces.

Limitations of the Waterfall model:

- Software Requirements are not really tested until a working system is available → errors in the SRS may be the last to be found → very costly to correct.
- Due to the late availability of a working system, a design or performance problem may go undetected until late in the testing phase → also costly to correct.

Where the SRS may contain components with identified significant risk factors, alternative models are preferred:

- The Throwaway Prototype model helps to resolve the first problem above - i.e. specifically designed to clarify user requirements (performed to a preliminary SRS).
- The Evolutionary Prototype model helps to resolve the second problem above - i.e. by creating subsets of the system that progress from prototype to working system. Allows performance measurements to be done early on critical components.

- The Spiral model provided a way of effectively combining throwaway prototypes, evolutionary prototypes and the Waterfall model by specifying an iterative approach made up of several cycles of the Waterfall model stages.

SOFTWARE DESIGN CONCEPTS FOR REAL-TIME SYSTEMS

Co-operation between concurrent tasks - primary problems encountered:

1. Mutual Exclusion - tasks require access to shared resources or devices (e.g. multiple readers/writers OS problem with the classical solution using binary semaphores).
2. Task synchronization - task co-ordination without the exchange of data (the solution is to use binary semaphores or event counters).
3. Producer/Consumer - tasks need to communicate and exchange data (the solution is to use intertask messaging which may be loosely or tightly-coupled).

Environments for Concurrent Processing

There are three main environments:

1. Multiprogramming - multiple tasks sharing one processor $\rightarrow$ virtual concurrency, i.e. the OS controls allocation of the processors to tasks.
2. Multiprocessing - multiple processors with shared memory $\rightarrow$ real concurrency with usually one virtual address space in which tasks execute and communicate.
3. Distributed processing - multiple nodes interconnected via a communications network $\rightarrow$ real concurrency with local address spaces with message passing.

SOFTWARE DESIGN METHODS FOR REAL-TIME SYSTEMS

Evolution of Software Design Methods:

- 1960's: minimal systematic analysis and design, some use of flowcharts, subroutines used for decomposition.
- 1970's: the growth of structured methods - top-down design, stepwise refinement. Two main approached developed - data flow orientated design (Structured Design - Demarco, 1978 and Gane, 1979 $\rightarrow$ lead on to Structured Analysis) and data structure design (Jackson Structured Programming - Jackson, 1975 and the Warnier/Orr method - Orr, 1977).
- late 1970's: for concurrent system design - introduction of the MASCOT notation - Simpson, 1979 - extension of the data flow approach that formalized intertask communication via channels and the specification of *pools* (encapsulated shared data structures). Data is accessed indirectly in the pools by calling access procedures that synchronize access.
- 1980's: Jackson System Development (JSD) - Jackson, 1983; Design Approach for Real-Time Systems (DARTS) - Gomaa, 1984; the Naval Research Labs software cost reduction method (NRL) - Parnas, 1984; Real-Time Structured Analysis and Structured Design (RTSAD) - Ward & Mellor, 1985 and Hatley & Pirbhai, 1988.
- late 1980's-1990's: emergence of object-orientated analysis and design (OOAD) methods - Booch, 1986 & 1991; Wirfs-Brock, 1990, Rumbaugh etal, 1991, Coad & Yourdon, 1992; Selic, Gullekson & Ward, 1995 (ROOM); Booch, Rumbaugh & Jacobson (UML), 1997; Douglass, 1998 (Real-Time UML).
- 2000's: convergence towards UML based approaches, e.g. Rational Unified Process, Booch et al, 1999; COMET (Concurrent Object Modelling and archectectural design mEThod Gomaa, 2000)

Requirements Specification versus Design Specification

Two views of a requirements specification are:

- the SRS should only address the external behavior of the system, i.e. the system is viewed as a "black box", e.g. the NRL method.
- the SRS should also be potentially executable → a prototype can then be developed from the specification → must include internal structure to support those requirements, e.g. the JSD method.

Most analysis methods, e.g. RTSA and OOA follow a problem-orientated approach, i.e. determine the problem domain components and the interfaces between them:

- RTSA methods map problem domain functions to functional modules in the design.
- OOA methods map problem domain objects to solution domain objects in the design.

In both RTSA and OOA, decisions made in the analysis stage often strongly influence the design, i.e. the problem orientated approach results in the scope of components being determined during analysis and their interfaces.

Criteria for Selecting Software Design Methods

1. The method must be in the published literature and not be proprietary.
2. The method must have been used on a real application.
3. The method must not be orientated to a specific language.
4. The method must be more than design notation, i.e. it must identify systematic steps to perform the design.

Real-Time Structured Analysis and Design (RTSAD)

Two main variations of RTSA have been developed: Ward & Mellor, 1985 and Hatley & Pirbhai, 1988. An Extended System Modelling Language (ESML) approach was an attempt (Bruyn etal, 1988) to merge the two approaches.

The primary addition over the conventional SADT approach is:
- state transition diagrams (STDs)
- control flows
- integration of STDs and DFDs via control transformations (in the Ward & Mellor approach) and control specifications (in the Hatley & Pirbhai approach).

The first phase of RTSA requires the development of :
- the essential model (Ward & Mellor)
- the requirements model (Hatley & Pirbhai)

This model has three views (the first two of which are most important for RT systems):
- the functional view
- the behavioural view
- the informational view (not supported in Hatley & Pirbhai)

Basic Terminology

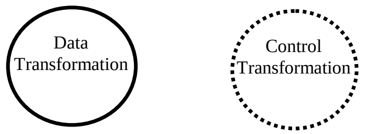
- *Functions*: basic elements the system is decomposed into (also called *transformations* or *processes*) which can be of the data or control type. The interaction between functions is in the form of data and control flows.
- *Modules*: during design the functions are mapped to modules.
- *Finite state machines*: in the form of state-transition diagrams (STDs) define the behavioural characteristics of the system. A control transformation is represented via a STD, decision tables or process activation tables.

- *Entity-Relationship (E-R) models*: used to identify the data stores and the relationships between them. While useful in data intensive applications they are not used in the Hatley & Pirbhai approach.
- *Module cohesion*: used in module decomposition to identify the strength or unity of a module.
- *Module coupling*: used in module decomposition to identify the degree of connectivity between modules.

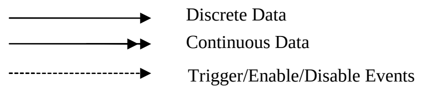
Notation (Ward/Mellor)

Data flow and control flow diagrams are the primary notation used in RTSA - they extend data flow diagrams to include event flows and control transformations.

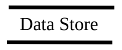
Transformations:



Flows:



Data Store :



Method (RTSA)

The method is iterative and not necessarily sequential:

1. Create the *system context diagram* - defines the boundary between the external environment and the system to be developed. The system is defined as a single data transformation with flows between sources and sinks of information at terminators on the diagram.

2. Perform data flow/control flow decomposition - the system transformation is structured into *functions* (or *processes* or *transformations*) with interfaces between them defined in terms of data or control flows:
   a) Hatley/Pirbhai: the emphasis is on hierarchical decomposition of function and data $\rightarrow$ multiple levels of data flow diagrams and data stores are specified with contents defined in a data dictionary.
   b) Ward/Mellor: the approach starts with an event list (set of system inputs) and specifies the system response to each event. The initial data/control flow diagram is non-hierarchical but subsequently is structured to decompose into lower-level diagrams if required.

3. Develop *Control Transformations* (Ward/Mellor) or *Control Specifications* (Hatley/Pirbhai):
   a) Ward/Mellor: input events trigger STD transitions through control transformations and output events are used to control execution of the data transformations. Control transformations may not be decomposed further.
   b) Hatley/Pirbhai: a control specification is defined by a STD, transition table or process activation table (which shows which processes are to be executed). Control specifications can be decomposed further.

4. Define *mini-specifications* (*process specifications*): usually defined in Structured English.

5. Develop *data dictionary*: define all data/event flows and data stores.

Method (RTSD)

In the design phase the Ward/Mellor and Hatley/Pirbhai approaches diverge; Hatley Pirbhai uses system architecture diagrams whereas Ward/Mellor continues with:

6. Allocate transformations to processors of the target system: possibly redraw the DFDs for each processor.
7. Allocate transformations to tasks: the transformations on each processor are allocated to concurrent tasks.
8. Structured Design: the transformations allocated to a task are structured into modules. The criteria for allocation is module coupling and cohesion coupled with the design strategies of Transform and Transaction Analysis.
   a) Module cohesion criteria: functional cohesion and informational cohesion are the strongest criteria.
   b) Module coupling: data coupling is the most desirable form of coupling (parameter passing between modules) whereas common coupling (shared global data) is the least desirable.
   c) Transform Analysis: a strategy to map a DFD to a structure chart diagram (SCD) using input/output flow $\rightarrow$ the input and output branches on the DFD are mapped to separate branches on the SC.
   d) Transaction Analysis: a strategy to map a DFD to a SCD by identifying the different transaction types $\rightarrow$ each transaction type on the DFD has a branch on the SCD with one controlling transaction center module.

Deliverables (RTSAD)

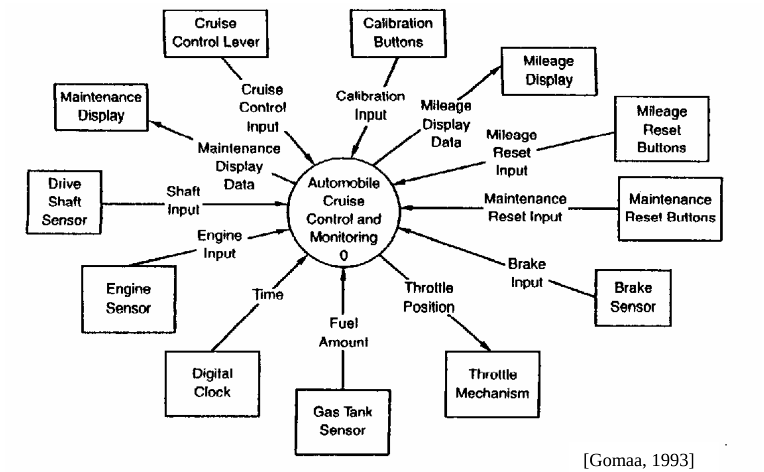
Analysis:
1. System Context Diagram
2. Hierarchical DFDs
3. Data dictionary
4. Mini-specifications of processes or transformations
5. STDs for each control transformation or specification

Design:
6. SCDs showing program decomposition into modules - a module is externally defined by input, output parameters and its function and internally defined by pseudocode.

Example: Automotive Cruise Control and Monitoring System

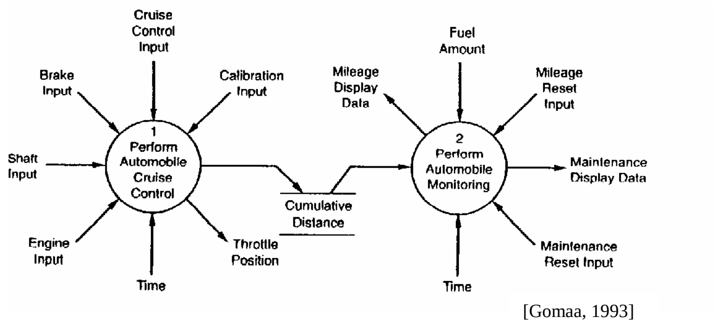
Structured Analysis:

1. Develop the System Context Diagram:



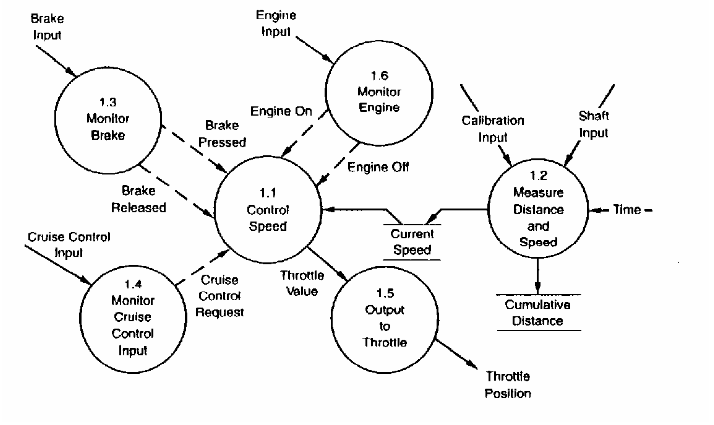
[Gomaa, 1993]

2. Decompose the System Context Diagram into major functions:



[Gomaa, 1993]

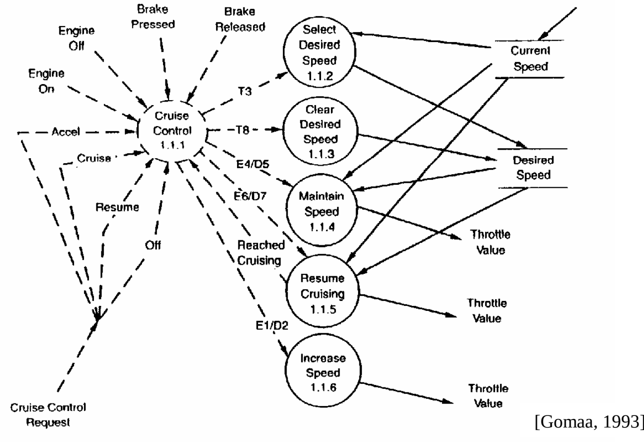
3. Perform Automotive Cruise Control DFD:



[Gomaa, 1993]

- Three transformations monitor sensors and generate appropriate events.
- A transformation computes the current speed and cumulative distance based on time and drive shaft input.
- A transformation computes the required throttle setting based on the current speed and cruise control setting
- A transformation converts the requested throttle value to the physical setting.

4. Control Speed DFD/CFD:

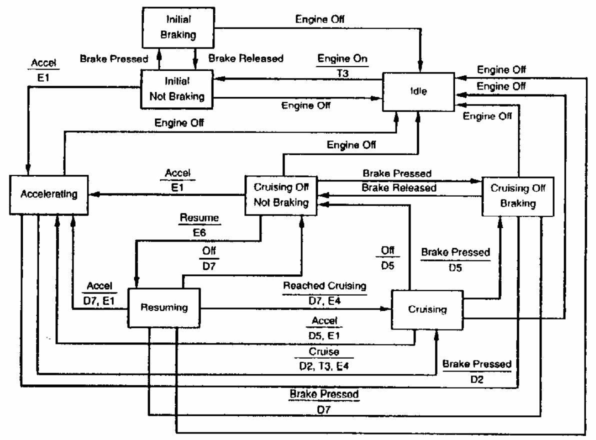


[Gomaa, 1993]

- The data transformation are triggered, enabled/disabled at state transitions:

E1 - Enable "Increase Speed" D2 - Disable "Increase Speed"
E4 - Enable "Maintain Speed" D5 - Disable "Maintain Speed"
E6 - Enable "Resume Cruising" D7 - Disable "Resume Cruising"
T3 - Trigger "Select Desired Speed"
T8 - Trigger "Clear Desired Speed"

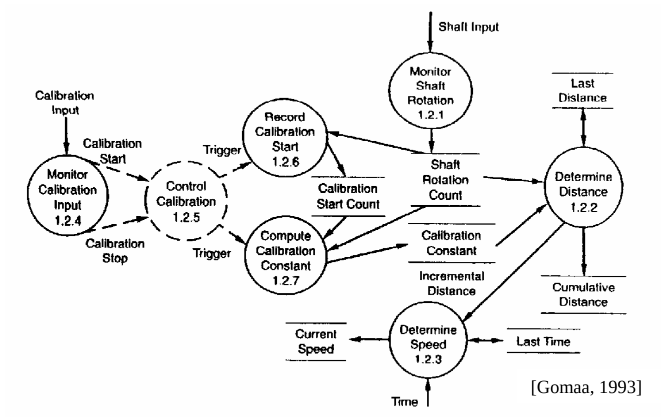
5. Cruise Control STD:



[Gomaa, 1993]

- The Control Speed DFD/CFD executes the Cruise Control STD. State transitions are labelled with input events/output events and the output event executes the corresponding data transformation on the Control Speed DFD/CFD.

6. Measure Distance and Speed DFD/CFD:



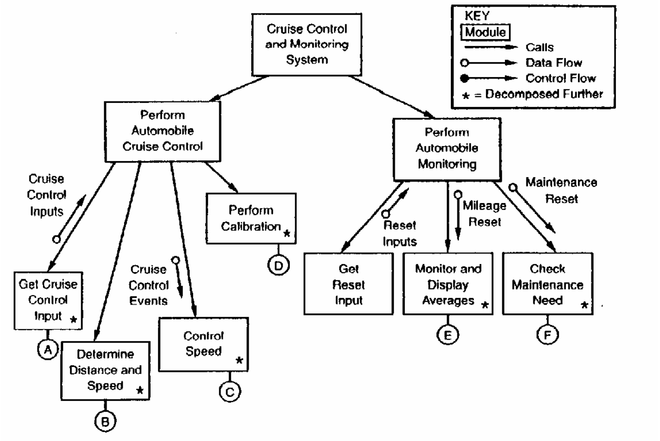
[Gomaa, 1993]

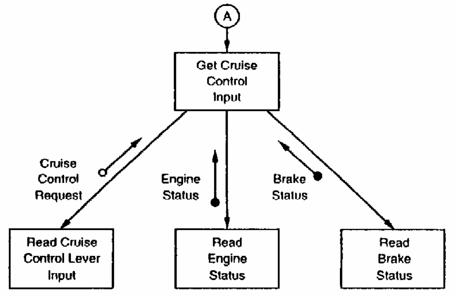
- Determine distance is periodically activated to compute the Incremental Distance travelled based on the current Shaft Rotation Count, Last Distance (last value of Shaft Rotation Count) and the Calibration Constant (the Shaft Rotation Count per km).
- The Incremental Distance is then added to the Cumulative Distance.
- Determine Speed receives the Incremental Distance and computes the Current Speed given the Incremental Time.

Structured Design:

Structured design provides no guidelines for decomposition into concurrent tasks $\rightarrow$ structure as one program initially.

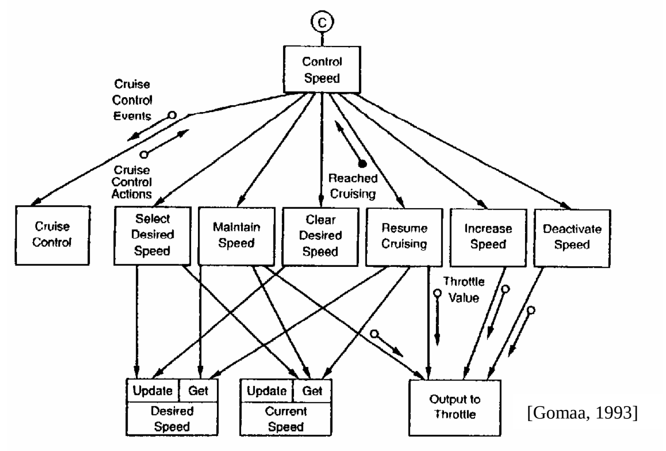
1. Perform Automobile Cruise Control:





[Gomaa, 1993]

- The four modules correspond to the four functions to be performed:
  - The Get Cruise Control Input is decomposed further into Read Control Lever Input, Read Engine Status and Read Brake Status.
  - Note that polled I/O is assumed, i.e. the inputs are polled on a cyclic periodic basis. For asynchronous I/O, i.e. interrupt driven, asynchronous I/O tasks should be assigned (but structured design provides no guidelines for this).



[Gomaa, 1993]

- Given a new cruise control input, the Control Speed module is called which then calls the Cruise Control module. This module encapsulates the Cruise Control transition table (with input events being used as the index into the table) and the output of the table comprises the Cruise Control Actions which is used by Control Speed to call the appropriate module.
- Although it would be possible to have common coupling for the Current Speed and Desired Speed data items, the Structured Design approach advises that Information Hiding Modules (IHMs) are be used to encapsulate these data stores $\rightarrow$ these also provide the operations used to access the data. For example, Maintain Speed calls Get Current Speed and Get Desired Speed, and based on the difference, sends the adjustment Throttle Value to the Output to Throttle module.

Timing Issues

- As the entire system is treated as a single sequential program, the timing aspects must be considered carefully. For example, assume that the control functions should be performed with an update rate of 100 msec and the monitoring functions should be performed with an update rate of 1 sec.
- A timer event can be used to awaken the Perform Automobile Cruise Control module every 100 msec and the Perform Automobile Monitoring is called every 10 activations of the timer event.
- The Perform Automobile Cruise Control loop calls Get Cruise Control Input, Determine Distance and Speed, Control Speed and Perform Calibration. In Control Speed the Select Desired Speed, Clear Desired Speed and Deactivate Speed modules are all run in response to events only.
- When the vehicle is in the following states:
  a) Cruising state - Maintain Speed is called periodically
  b) Accelerating state - Increase Speed is called periodically
  c) Resuming state - Resume Cruising Speed is called periodically.
- The Get Cruise Control Input module and Output to Throttle must be concurrent or interleaved in a sequential program.

Assessment of Method

- It was the major RTSAD method with many successful applications.
- Wide range of CASE tools to support the method.
- Minimal guidance on system decomposition.
- Structured Design does not specifically address task structuring.
- Application of Information Hiding methods is minimal - better in NRL method and OOD method.

Design Approach for Real-Time Systems (DARTS)

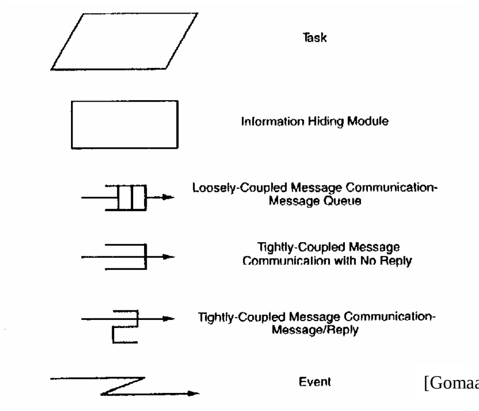
This approach (due to Gomaa, 1984-1987) emphasizes the decomposition of the real-time system into concurrent tasks and defining the interfaces between the tasks → it provides task structuring criteria and guidelines for defining the task interfaces.

Basic Concepts:

- *Task structuring criteria*: a set of heuristics derived from experience - the criteria are applied to the transformations (functions) on the DFD/CFDs developed using RTSA based on the temporal sequence in which the functions are executed.
- *Task interfaces*: provided in the form of message communication, event synchronization or information hiding modules (IHMs).
- *Information hiding*: used for encapsulating data stores and state transition tables.
- *Finite state machine*: defined in the form of STDs
- *Evolutionary prototyping* and *incremental development*: assisted by the identification of *system subsets* using *event sequence diagrams*, i.e. the sequence of tasks and modules to process an external event is identified so that they may be incrementally developed.

Notation:

- DFD/CFDs and STDs from RTSA are extended to include event flows. SCDs from RTSD are also used to show task decomposition into modules.
- *Task architecture diagrams* (TADs) are used in DARTS to show the decomposition of the system into concurrent tasks and their interfaces.


Task
Information Hiding Module
Loosely-Coupled Message Communication-Message Queue
Tightly-Coupled Message Communication with No Reply
Tightly-Coupled Message Communication-Message/Reply
Event

[Gomaa, 1993]

Method (DARTS)

1. Develop system specification using RTSA.
2. *Structure the system into concurrent tasks*: these criteria are applied to the lowest level of the hierarchical set of DFD/CFDs. A preliminary TAD is drawn using the task structuring criteria:
    - I/O data transformations that interface to external devices are mapped to asynchronous I/O tasks, periodic I/O tasks, or resource monitor tasks.
3. *Define task interfaces*:
    - Data flows between tasks are mapped to either loosely-coupled or tightly-coupled message interfaces.
    - Event flows are mapped to event signals.
    - Data stores form the basis of IHMs.
    - A timing analysis can be performed using event sequence diagrams.
4. *Design each task*:
    - Each task represents a sequential program which is structured into modules using structured design - either transform analysis or transaction analysis can be used for this purpose.

- The function of each module is defined and then the internals of each module are designed.

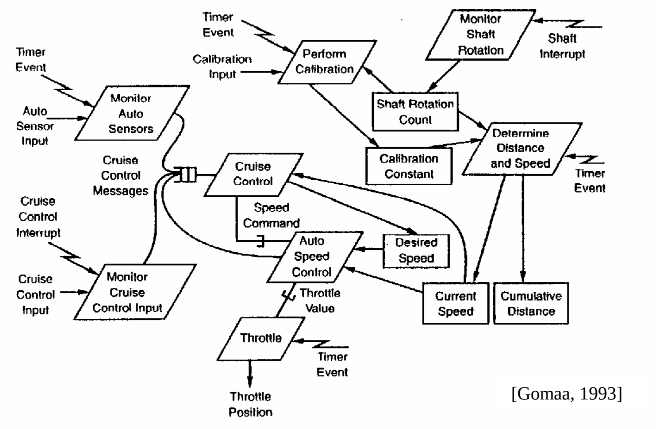
Deliverables (DARTS)

1. RTSA specification
2. Task structure specification - defines the concurrent tasks in the system (function and interfaces).
3. Task decomposition - structure of tasks into modules (function, interfaces and detailed design in PDL).

Example: Automotive Cruise Control and Monitoring System

The RTSA specification follows from the analysis phase of the RTSAD method.

Task Structuring:



1. As shown above the Cruise Control subsystem can be represented as TADs . The Cruise Control subsystem is decomposed into asynchronous device input tasks of Monitor Cruise Control Input and Monitor Shaft Rotation. Each task is activated by an external interrupt.

2. Monitor Auto Sensors and Perform Calibration are both periodic input tasks - temporally cohesive (since both are linked to the same timer event).
3. Cruise Control is a high priority control task $\rightarrow$ executes the STD
4. Auto Speed Control is a task that is sequentially and functionally cohesive (since all its functions are related to speed control and the are all constrained to execute sequentially).
5. Throttle is a periodic output task.
6. Determine Distance and Speed is a periodic sequentially cohesive task which computes the Cumulative Distance and Current Speed at regular intervals.

Task Interfaces:

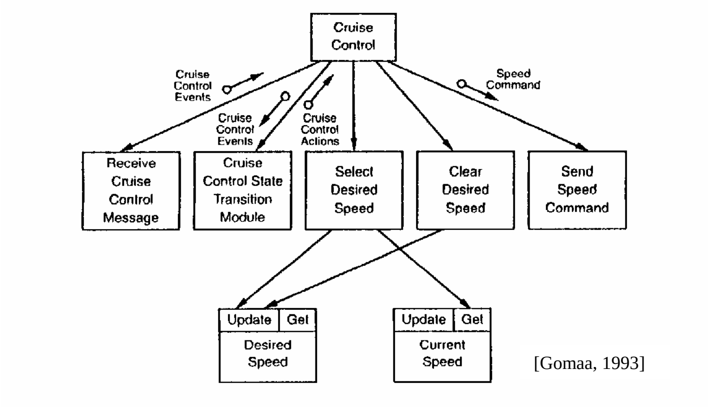
All tasks communicate via messages or IHMs:

1. Data stores that are accessed by more than one task are mapped to IHMs. Access control is achieved via semaphores.
2. The interface to the Cruise Control task is via a loosely- coupled FIFO queue $\rightarrow$ ensures that I/O tasks are not held up by Cruise Control, and if input events arrive in quick succession none are lost.
3. The interface between Cruise Control and Auto Speed Control is tightly coupled with no reply.

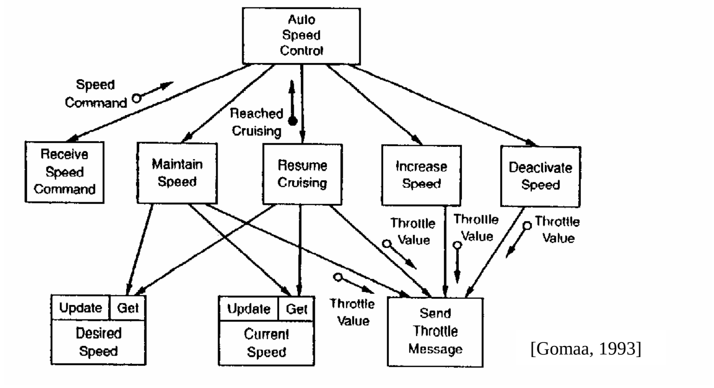
Structured Design:

Given the task interface design, the next step is to design each task (which represents a sequential program) $\rightarrow$ can use the Structured Design method to decompose into modules.

The Cruise Control Task has a main procedure that is (by convention) given the same name as the task.



1. Cruise Control calls Receive Cruise Control Message to wait for an input message (i.e. the task is suspended on this input).
2. Cruise Control calls the STM module passing it the appropriate event and returning the appropriate action from the transition table.
3. Where the action is Select Desired Speed or Clear Desired Speed, the appropriate procedure is called.
4. Other actions are sent as speed commands to Auto Speed Control.
5. There are two IHMs, Current Speed and Desired Speed - Update and Get ensure synchronized access to the data items.



1. Auto Speed Control is suspended on Receive Speed Command waiting for a message.
2. Transaction Analysis can be applied to this task by identifying the command received as a transaction, Auto Speed Control is the transaction processor which calls the appropriate transaction handling procedures.

Assessment of Method

- Emphasize decomposition of system into concurrent tasks and provides criteria for identifying tasks.
- Gives guidelines for interfaces between tasks.
- Emphasizes use of STDs and provides a transition from RTSA to a real-time design by providing the decomposition principles.
- Although IHMs are used, they are not as extensively employed as in OOD methods.
- A potential problem with DARTS is that it is dependent on a well performed RTSA phase but the RTSA approach is weak on system decomposition guidelines. To minimize the impact of this limitation, the DARTS approach specifies that attention should be directed to control flow (i.e. STDs) before data flow.

Object-Orientated Design for Real-Time Systems

OOD was initially a design method based on the primary concepts of abstraction and information hiding. The two main approaches to OOD in the literature initally diverged on the importance of inheritance:

1. Inheritance is viewed as a desirable but not essential feature of OOD - the view taken by the Ada programming community [Booch].
2. Inheritance is viewed as an essential feature of OOD - the view taken by the OO programming community - e.g. Smalltalk [Goldberg] and Eiffel [Meyer].

The initial advantage of Booch's approach (1986) was that it was more applicable to concurrent and real-time systems design – it supported objects through information hiding but not classes or inheritance, Booch’s later approach (1991) supported classes and inheritance.

Notation:

- *Class diagrams* - shows the system classes and the inheritance and uses relationships between all classes.
- *Object diagrams* - shows the system objects and the relationships between all objects.
- *State transition diagram* - shows the object states and the events that cause the transitions between object states.
- *Timing diagrams* - shows the dynamic interaction between objects by showing the time-ordered sequence of execution of operations between objects.
- *Module diagrams* - shows the allocation of classes and objects to modules in the physical system.
- *Process diagrams* - shows the allocation of concurrent processes (tasks) to processors in the physical system.

Steps in Method:

Booch referred to his OOD method as "Round-trip Gestalt Design" $\rightarrow$ a highly iterative holistic approach.

1. *Identify the classes and objects* - find the key abstractions in the problem domain. Booch has had three attempts at this:
   a) Identify objects by finding all nouns and all operations by finding all verbs in the specification.
   b) Use Structured Analysis and identify objects from the DFD, i.e. sources or sinks of data have corresponding software objects to "hide them".
   c) Directly analyse the problem domain and apply object structuring criteria, i.e. each entity has a class which has defined attributes and its relationships with other classes are established.

2. *Identify class and object semantics* - the interface and operations of each object is determined. This is very iterative due to the effect of a change in one object's interface on another object's definition.
   - Preliminary class and object diagrams are developed.

3. *Identify the relationship between classes and objects* - closely coupled to the above step - static and dynamic dependencies between objects are determined, i.e. object visibilities are considered.
   - Inheritance and uses structures are defined.
   - Class and object diagrams are refined.
   - Preliminary module diagrams are developed.
   - Object classification - as *server* (only provide operations), *actor* (only use operations) or *agent* (both provide and use operations).

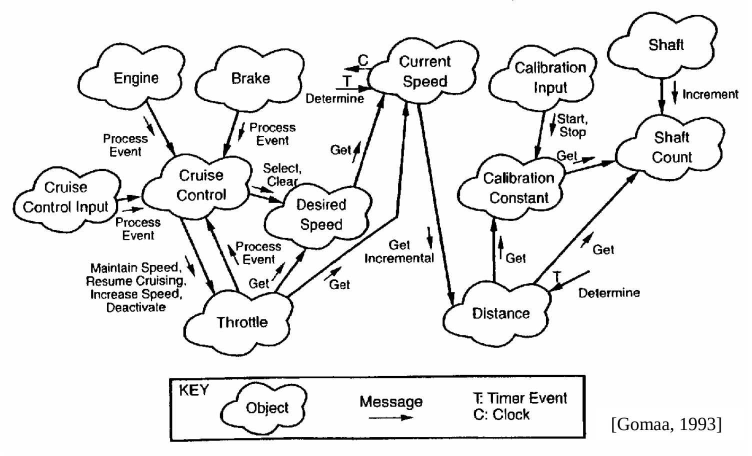
4. *Implement classes and objects* - classes and objects are allocated to information hiding modules and programs are allocated to processors. Object internals are designed and developed.

Deliverables

1. Class diagrams and class specification.
2. Object diagrams and object specification.
3. STDs and timing diagrams.
4. Module diagrams and module specifications.
5. Process diagrams and process specifications.

Example: Automotive Cruise Control and Monitoring System

Preliminary Class and Object diagrams are developed:



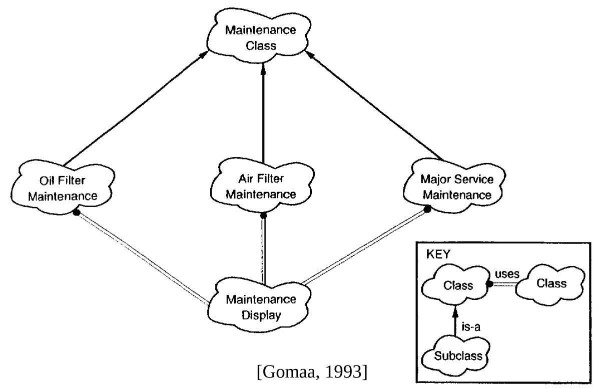
[Gomaa, 1993]

Note that the objects Engine, Brake, and Cruise Control Input (which would be classified as *actors*) send messages to Cruise Control corresponding to input from external devices they encapsulate. Cruise Control (which is classified as an *agent*) encapsulates the STD and invokes operations in other objects, e.g. it sends a message to Throttle to Maintain Speed, Resume Cruising, etc. Shaft Count is an example of a *server* object.

Identifying Relationships between Classes and Objects

The static and dynamic dependencies between objects are determined and the inheritance and uses structures are defined.

In this example all classes have only one instance (the object of the same name), except for the Maintenance class which has subclasses of Oil Filter Maintenance, Air Filter Maintenance, Major Service Maintenance. Each of the subclasses uses the same Maintenance Display class.
The class diagram is very simple for this example:



[Gomaa, 1993]

Assessment of Method:

1. The method is strongly based on information hiding, classes and inheritance - key concepts in OO software design.
2. The structuring of the system into objects makes the system more maintainable and the components potentially reusable.
3. The provision of inheritance allows components to be modified in a controlled manner as required.
4. Maps well to language that support information hiding modules (e.g. Ada and Modula-2) and to languages that support classes and inheritance (e.g. C++, Smalltalk and Eiffel).
5. Doesn't really address task structuring.
6. A highly iterative procedure is required and procedures at each step are not very specific.
7. Although the method does provide for timing diagrams it does not fully address timing constraints.

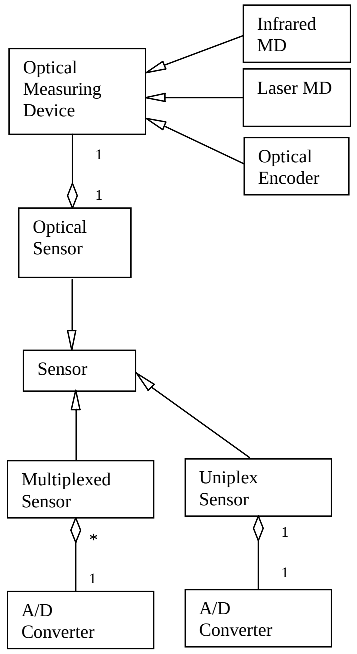
Real-Time UML

The purpose of UML is to support an integrated OO design methodology, and because of its completeness and widespread support it has been extended to model real-time embedded systems. Our focus here, while briefly reviewing the primary notation of UML, is the extensions to support real-time modelling. - in particular, its support for temporal scenario modelling and representations of tasking.

UML Class Diagrams

Relations among classes and objects:

- Association - messaging between objects bidirectional or unidirectional (open arrow)
- Aggregation - one object containing another (diamond indicates owner)
- Composition - aggregation plus create/destroy role (closed diamond or shown by inclusion)
- Generalization - inheritance of characteristics of a parent class (shown by open arrow directed towards parent)
- Refinement - generic (or template) elaborations of incomplete class specifications (dashed lines with closed arrows)

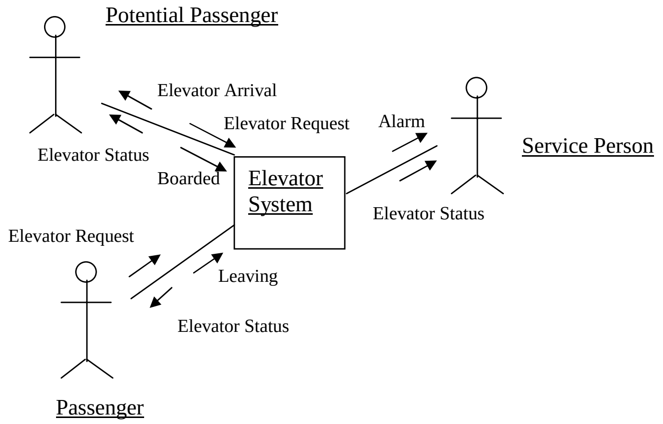


Instance multiplicity:

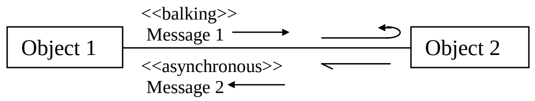
- Integer - number of objects participating in the relationship
- * - unspecified multiplicity greater than zero

UML Object Diagrams

The UML object diagram shows the relationships between objects in the system, and although a context diagram is not explicitly supported, appropriate stereotypes on objects and messages can be used to create an object context diagram, e.g. for the elevator controller:



Message Stereotypes



The context-level flows or *messages* are abstract at this point of the design, but expected data content, arrival pattern and synchronisation pattern can be defined (implementation being deferred until later in the design process).

UML defines events as messages with a class stereotype of <<signal>> and attributes:

- Message arrival - *episodic* (unpredictable but bounded by a *minimum interarrival time*, may be random or bursty) or *periodic* (characterised by a *period* and *jitter*).
- Message synchronisation - *call* (sender blocked but single threaded), *waiting* (sender blocked but multi-threaded) and *asynchronous* (non-blocking) [Booch added extensions of *balking* (sender aborts if receiver not ready) and *timeout* (balking with a waiting time) synchronisations].

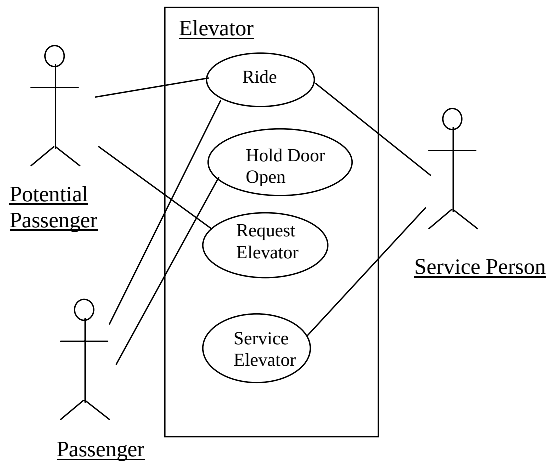
Use Cases

A *use case* is used to capture a customer requirement and it shows the general cases of interaction between the system and all external objects - it is built from the underlying event flows on the context diagram.

Ultimately use cases may be decomposed into *scenarios* which show the detailed object interactions (i.e. scenarios are instances of use cases). Use cases may also be abstracted from a number of scenarios defined through discussion with the customer.

Some example scenarios from the *Ride* use case:

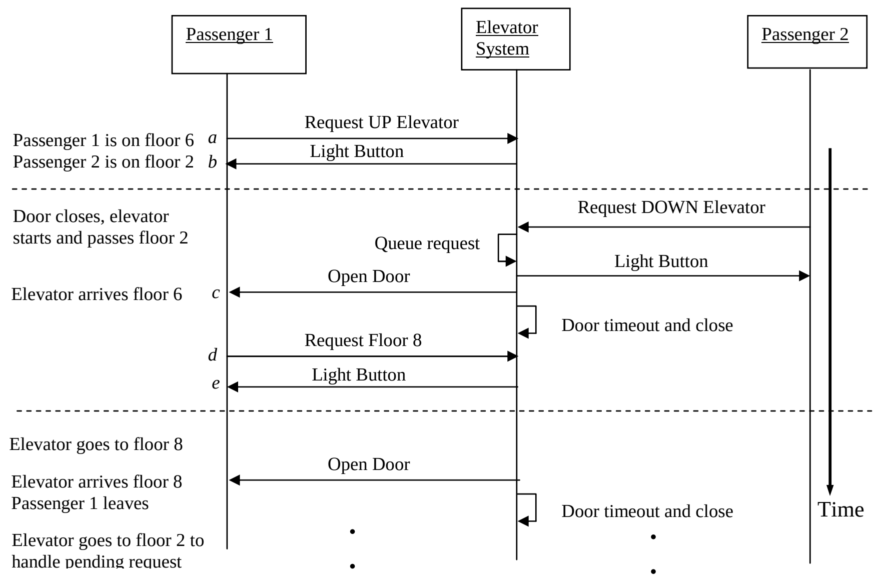
- Elevator at floor
- Elevator must travel to the floor
- An elevator must handle a pending request (before or after picking up a potential passenger).
- Passenger issues another request



Scenarios can be modelled in two ways in UML: *sequence diagrams* and *collaboration diagrams*. Sequence diagrams (the most commonly used) emphasise messages and their sequence, while collaboration diagrams tend to emphasise system object structure.

Vertical lines are used to represent objects, and horizontal directed lines represent messages. Each message starts from an originator object and ends at a target object, with time flowing top to bottom. The text annotations describe object names, message names and the conditions associated with the message.

UML Sequence Diagrams



Additional information on sequence diagrams

- Event identifier: reference character for message triggers and outcomes (e.g. *a*: Passenger 1 is on floor 6, *b*: Elevator arrives on floor 6).
- Timing constraints - two types:
  1. Marker bar with time difference between events, e.g. 20ms ├─┤
  2. Relational expressions between events, e.g. $\{e - d < 500 \text{ ms}\}$

UML currently only provides limited specification of timing constraints via OCL, so additional notation is usually required, e.g. a soft time constraint between two events could be represented by:

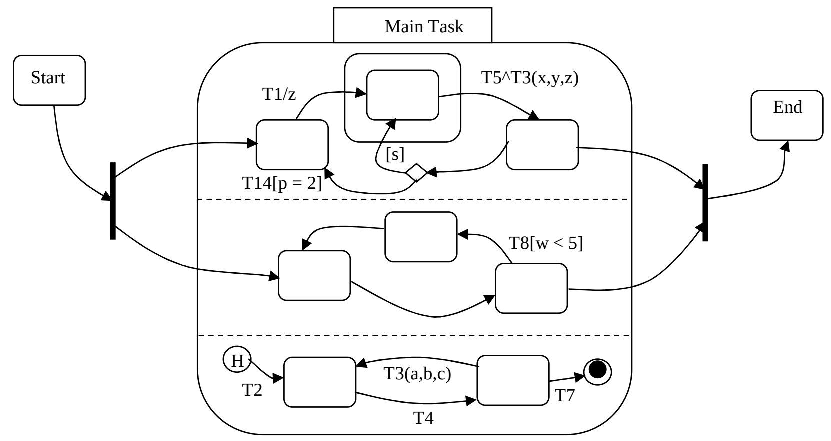
AVE $(b - a) <= 3$ sec where AVE is an average operator.

There are also variations on core UML that bridge the gap between sequence diagrams and state diagrams by allowing *state marks* on sequence diagrams, i.e. rounded boxes on the vertical object lines that represent the change of state of an object in response to events.

UML State Diagrams

UML state diagrams extend the basic concept of Finite State Machines in three ways:

- Nested hierarchy, e.g.
- Concurrency, e.g. ---------
- Extended transitions, e.g.
  event-name(parameters)[guard]/action list^event list



State diagrams, although capturing all state dependent behaviour of an object, do not show typical paths through the system - this is the function of scenarios. Scenarios can be represented by conventional timing diagrams (not defined in UML) or sequence diagrams (which are defined in UML).

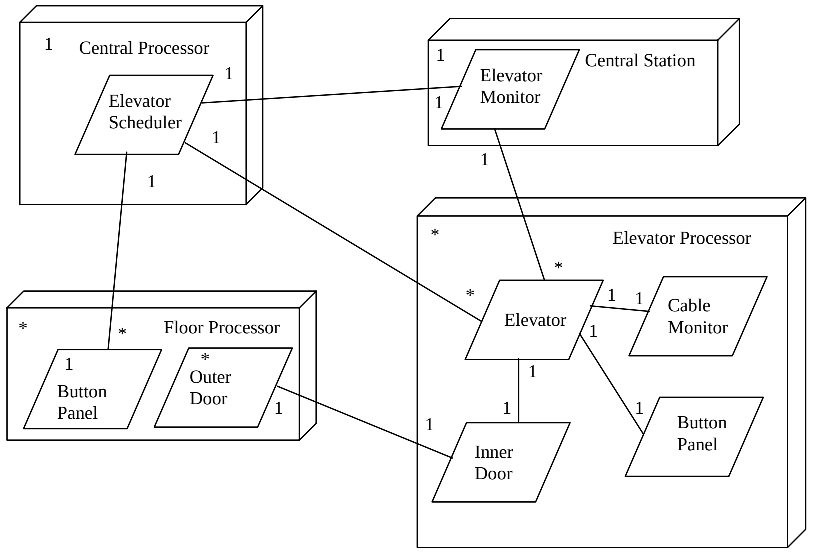
Events can also be structured in object hierarchies, e.g. input events from a mouse or keyboard, or exception event objects handling progressively more detailed exceptions.

UML System Task Diagrams

UML can model concurrency in two ways:

- Class and object diagrams can show tasks directly
- State diagrams can show concurrent component execution

A System task diagram is a filtered object diagram than only shows concurrent threads of execution. An extension to core UML created for the task object is the <<active>> stereotype.



Each processor is identified with its multiplicity (1, 2, …, *) and the multiplicity of each task is shown where appropriate. Each task is rooted in a single active composite object that receives events for that task and dispatches them to the appropriate object within the task.

N.B. There are many extensions to UML in preparation (V1.4 was released in late-2001, V1.5 is in draft and V2.0 is in preparation), see www.omg.org/uml for the latest.

Rational Rose Real-Time (formally ObjectTime)

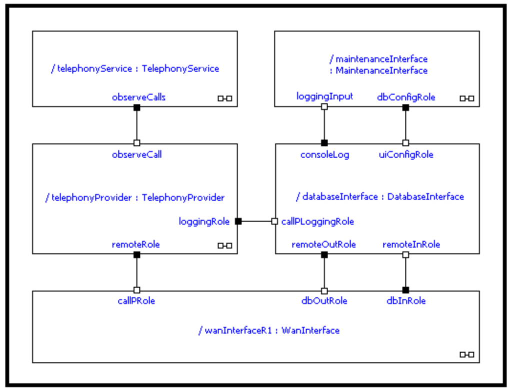
UML for Real-Time grew out of the Real-Time Object-Orientated Modelling langauge (ROOM) with terminology being aligned with UML 1.1 but retaining all the semantics of ROOM models. There are a number of specific additional constructs used in Rose Real-Time that accommodate the mapping from ROOM models.

Capsules

UML for Real-Time provides built-in light weight concurrent objects, known as *capsules*. Capsules are simply a pattern for providing light-weight concurrency directly in the modeling notation. A capsule is implemented in Rose RealTime as a special form of class.

Capsules are highly encapsulated and communicate through special message-based interfaces called *ports*. The ports are in turn connected to other capsules, enabling the transmission of messages among capsules. The advantage of message-based interfaces is that a capsule has no knowledge of its environment outside of these interfaces, making it a much more flexible and robust than regular objects.

Capsule structure diagrams


/ telephonyService : TelephonyService
observeCalls
/ maintenanceInterface : MaintenanceInterface
loggingInput
dbConfigRole
observeCall
/ telephonyProvider : TelephonyProvider
loggingRole
remoteRole
consoleLog
uiConfigRole
/ databaseInterface : DatabaseInterface
callPLoggingRole
remoteOutRole
remoteInRole
callPRole
dbOutRole
dbInRole
/ wanInterfaceR1 : WanInterface

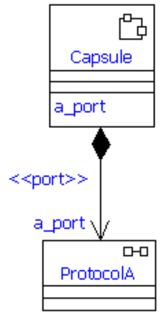
A new diagram has been introduced that specifies the capsule's interface and its internal composition. The diagram is called a capsule structure diagram (based on the UML 1.3 specification collaboration diagram). The semantics of the capsule structure diagram allow Rose RealTime to generate detailed code to implement the communication and aggregation relationships among capsules.

Mapping capsules to threads

Rose RealTime allows designers to make use of the underlying multi-tasking operating system so that the processing of a capsule on one thread does not block the processing of capsules on other threads. Designers can specify the physical operating system threads onto which the capsules will be mapped at run-time. Not every capsule should run on a separate thread. For most capsules, it is sufficient to leave them in one thread and allow the Services Library controller to invoke their behavior as messages arrive.

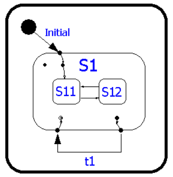
Capsules with transitions that may block, or that have excessively long processing times, should be placed in separate threads. Deciding which capsules need to execute in different threads is a design issue.

Protocols



The set of messages exchanged between two objects conforms to some communication pattern called a *protocol*. A protocol comprises a set of participants, each of which plays a specific role in the protocol. Each such protocol role is specified by a unique name and a specification of messages that are received by that role as well as a specification of the messages that are sent by that role. As an option, a protocol may have a specification of the valid communication sequences; a state machine may specify this. Finally, a protocol may also have a set of interaction sequences (shown as sequence diagrams). These must conform to the protocol state machine, if one is defined.

Hierarchical State Machines

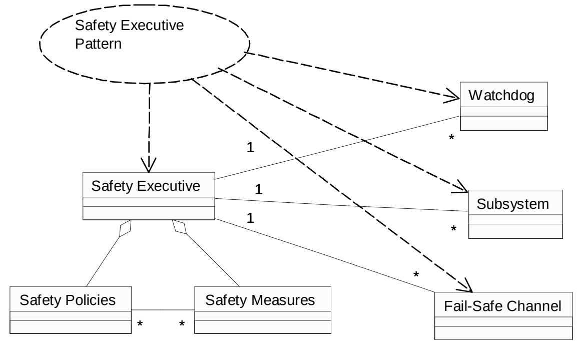


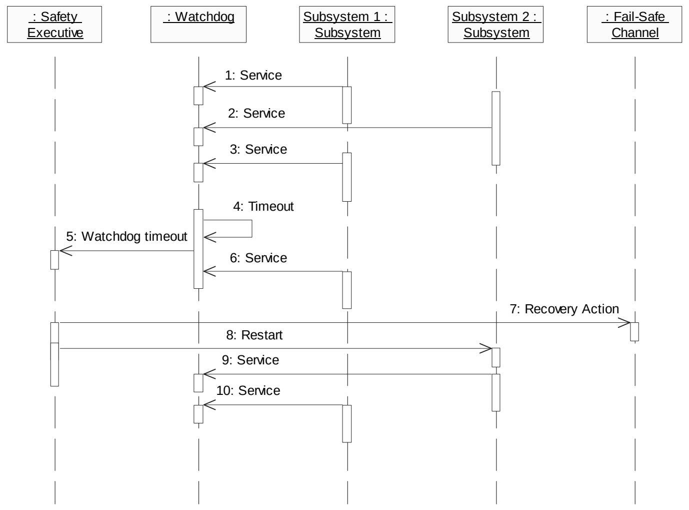
A state can be composed of other states, called substates. This allows modeling of complex state machines by abstracting away detailed behavior into multiple levels.
A state that has substates is called a composite state.

UML Patterns for Real-Time Software Design

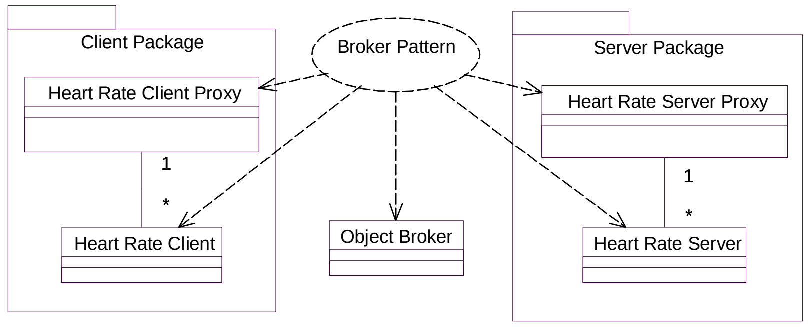
Design patterns are simply a formalisation of a particular approach to a common problem in a context. They can be described by text (usually in a format that specifies the problem context, the solution and any constaints) and/or diagrams. UML is particularly useful in this context. Two examples will help illustrate (note that in UML notation a pattern is described by a dashed oval connected to partcipating entities with a dashed line).

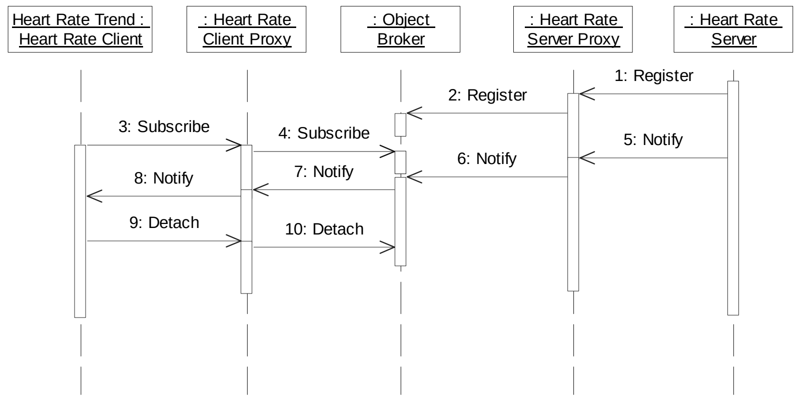
Safety Executive Pattern





Broker Pattern


Client Package
Broker Pattern
Server Package
Heart Rate Client Proxy
Heart Rate Server Proxy
1
*
1
*
Heart Rate Client
Object Broker
Heart Rate Server


Heart Rate Trend : Heart Rate Client
: Heart Rate Client Proxy
: Object Broker
: Heart Rate Server Proxy
: Heart Rate Server
1: Register
2: Register
3: Subscribe
4: Subscribe
5: Notify
6: Notify
7: Notify
8: Notify
9: Detach
10: Detach