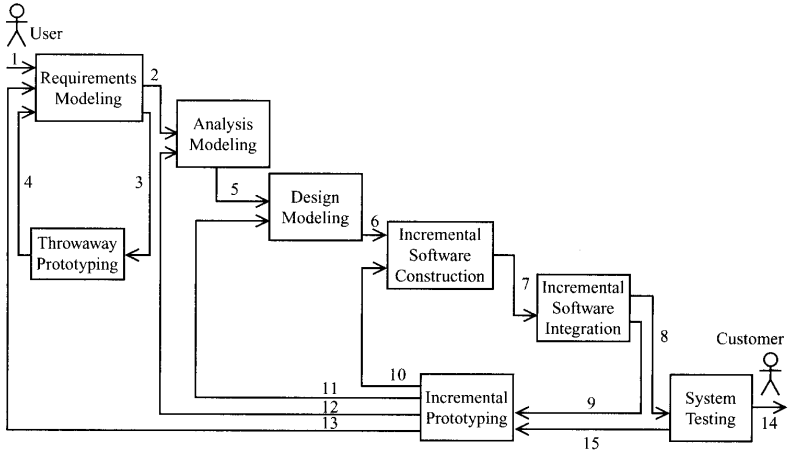
REAL-TIME DISTRIBUTED SYSTEMS DETAILED DESIGN METHODOLOGY

We consider one methodology in detail that combines many of the best practice elements of the RTDS design methodologies developed over the last two decades.

COMET (Concurrent Object Modelling and architectural design mEThod, Gomaa 2000)

A highly iterative OO software development process based on use cases:



[Gomaa, 2000]

- Requirements Modelling: functional requirements are defined in terms of use cases - can use throwaway prototyping to refine.
- Analysis Modelling: static (structural relationships between classes) and dynamic models (interactions between objects and state dependent behaviour) are developed.

- Design Modelling: software architecture describes the subsystems and how they are structured and interact. For concurrent systems the emphasis is on tasking in addition to OO implementation issues.
- Incremental Software Construction: system subset selection for construction by use cases which includes analysis, design and unit test.
- Incremental Software Integration: integration testing between units (focussing on interfaces) which forms an incremental prototype (iterate if problems).
- System testing: functional testing against specifications.

All these steps are very comparable with:

- Unified Software Development Process (Rational)
- Spiral Model (Boehm).

Focus on the elements of the process that distinguish RTDS methodologies from standard Software Engineering approaches:

- Highly similar in the requirements modelling and static analysis modeling phases
- Variations and change in emphasis shows up in dynamic modelling and architectural modeling phases

Dynamic Modelling

In an OO design approach, Dynamic (or Behavioural Modelling) emphasis is on the inter-object interactions or intra-object interactions via state-dependent execution.

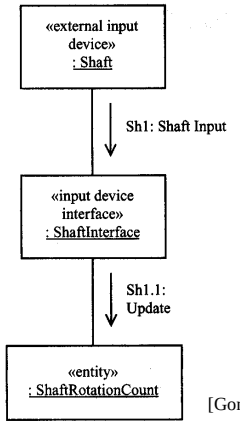
In UML, interaction diagrams (collaboration or sequence diagrams) focus on messages between objects, and dynamic analysis is used to construct models of the message passing between objects.

Messages can be seen as an event plus message or event attributes which are passed between objects (and then used as events to initiate state-dependent transitions if appropriate for that object).

message = event (message attributes)

Investigating the sequence of message passing or events has been referred to as **event sequence analysis** in previous approaches.

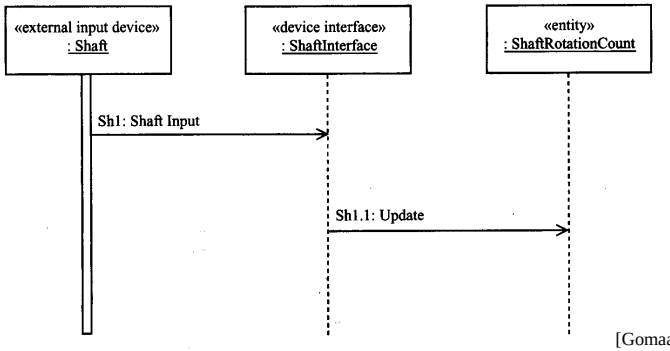
Collaboration (or Sequence) diagrams are developed from use case scenarios and show all message passing and interacting objects, e.g:



[Gomaa, 2000]

Stereotypes are used to distinguish object (or class) types.

Similar information can be shown on a sequence diagram:



[Gomaa, 2000]

Of note, in the analysis phase no assumptions about mapping messages to object operations are made or when objects are active or inactive. Different approaches place a different emphasis on the different diagrams, e.g. COMETs preference is for collaboration diagrams to aid in the architectural design phase.

Message labels can be further complicated by the inclusion of sequence numbers, recursion indicators and clauses and condition clauses:

[message sequence number] [recurrence indicator] [recurrence clause] [condition clause] message name [(message arguments)]

e.g. 1*[j:=1,n] [x > 0] message1(argument1, ...)

The message sequence number can optionally be hierarchical and include letters to indicate the use case it is associated with.

A **message sequence description** may also be developed to supplement the collaboration or sequence diagram to detail what messages are passed between objects and what receiving objects do when the various messages arrive in the indicated sequence, e.g. identify object attributes that are accessed.

Dynamic Analysis

The purpose of dynamic (or behavioural) analysis is to determine how the objects interact with each other to perform the use cases. This analysis might show a need for additional objects and/or interactions. Dynamic analysis may either be non-state-dependent or state dependent, i.e. outputs may depend on more than just actor inputs but prior events. For RTD systems state-dependent analysis is typically required, but consider the simpler type first:

Non-state-dependent dynamic analysis:

1. Determine interface objects – identify the actors initiating the use case and the external interface the interface object interacts with.
2. Determine internal objects – identify all internal control and entity system objects participating in the use case.
3. Determine object collaboration – identify all interactions between internal objects and develop the collaboration or sequence diagrams.
4. Consider alternative sequences – exception or error handling.

State-dependent dynamic analysis:

1. Determine interface objects.
2. Determine state-dependent control object – responsible for executing the state diagram.
3. Determine other internal objects.
4. Determine object collaboration.
5. Determine state diagram execution
6. Consider alternative sequences

The main difference is that messages arriving at a state-dependent control object causes state-transitions on the statechart, which then generates actions which either may become events (hence messages) for other control objects, or actions/activities for the initiating control object.

State diagram execution may be based on an earlier determined statechart which in the dynamic analysis phase needs to be refined and/or confirmed.

State Diagram Development

Given typical scenarios from use cases developed in the requirements modeling phase, each external event in the scenario causes a transition to a new state. Associated with that transition may be some actions, and associated with each state may be some activities on entry, exit, etc.

Each event sequence through the scenario should be able to be traced through the execution of the developing statechart. All alternative event sequence paths corresponding to all scenarios of the use case are considered.

Example: Cruise Control system – Control Speed use case state diagram development

Actor: Driver
Summary: This use case describes the automated cruise control system, given the driver inputs via the cruise control lever, brake, and engine external input devices.
Precondition: Driver has switched on the engine and is operating the car manually.
Description:
A typical abbreviated scenario is:

1. Driver moves the cruise control lever to ACCEL and holds it there, and the system initiates automated acceleration.
2. Driver releases the cruise control lever to cruise at constant speed, and the system starts maintaining the speed of the case at cruising speed which is stored for future reference.
3. Driver presses the brake to disable cruise control so that it is then under manual control.
4. Driver moves the cruise control lever to RESUME to resume cruising at the previously stored cruising speed.
5. When the system detects cruising speed has been reached, it stops automatic acceleration (or deceleration) and maintains cruising speed.
6. Driver moves the cruise control lever to the OFF position which disables cruise control and returns the car to manual operation.
7. Driver stops the car and switches of the engine

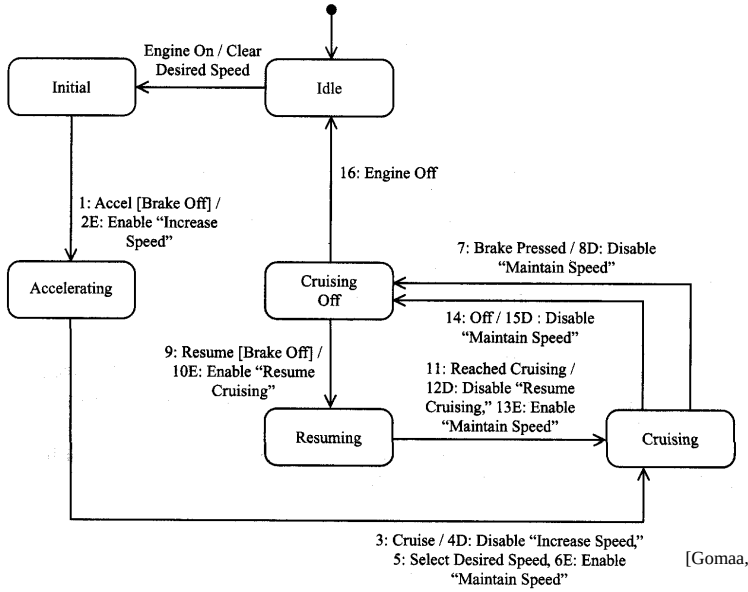
Alternatives:
The following are the alternative external input events:

1. Cruise control lever events: ACCEL, CRUISE, RESUME, OFF (where the CRUISE position is the default central position of the lever).
2. Brake system events: Brake Pressed and Brake Released (automated cruise control is only possible after the brake has been released).
3. Engine system events: Engine On and Engine Off (engine off disables any automated control system activity).

Postcondition: Car is stationary with the engine off.

Statechart Development

The *Idle* state is the initial statechart state, and the *Initial* state is reached when the engine is switched on (which is the precondition of the use case). The following external events then occur in the scenario which are mapped to the statechart shown below:



[Gomaa, 2000]

1. ACCEL cruise control event – providing the brake is off (which becomes a condition) a transition is made to the *Accelerating* state to accelerate the vehicle so an *Increase Speed* activity is enabled.
2. CRUISE cruise control event - the cruise control lever is released which cause a transition to the *Cruising* state. This also initiates the following actions:
   - automatic acceleration stops, so the *Increase Speed* activity is disabled
   - the cruising speed is stored through the *Select Desired Speed* action
   - vehicle speed is maintained through enabling the *Maintain Speed* activity
3. BRAKE pressed – the *Maintain Speed* activity must be disabled and a transition is taken to the *Cruising Off* state.
4. RESUME cruise control event – providing the brake is off a transition is made to the *Resuming* state and the *Resume Cruising* activity is enabled (which initiates acceleration or deceleration towards the previously stored cruising speed).
5. When the *Resuming Cruising* activity has reached the stored cruising speed, this activity can be disabled, the *Maintain Speed* activity can be enabled, and the *Cruising* state is entered.
6. OFF cruise control lever event – a transition to the *Cruising Off* state and the *Maintain Speed* activity is disabled.
7. ENGINE off – the *Idle* state is re-entered.

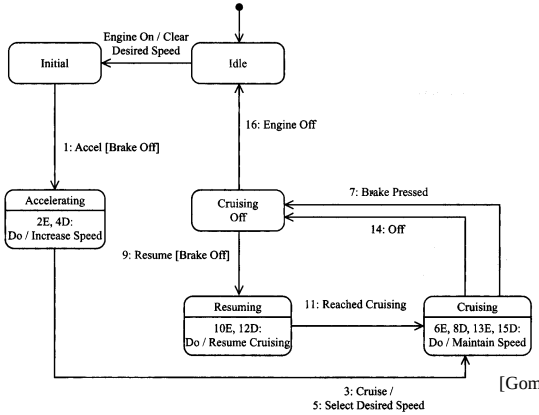
Note that almost all of these transitions are caused by the driver's actions so they are directly visible (apart from the transition from the *Resume* state to the *Cruising* state).

Statechart Refinement

Given the initial statechart corresponding to the Control Speed use case, a number of refinements can be made:

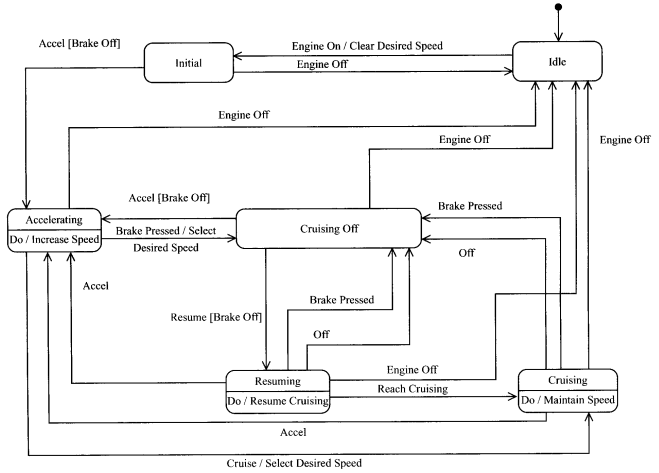
1. Enables and Disables of activities on transitions into states and out of states can be shifted to the states, e.g. The *Increase Speed* activity is enabled on entry to the *Acclerating* state and disabled on exit.

This results in the following statechart:



[Gomaa, 2000]

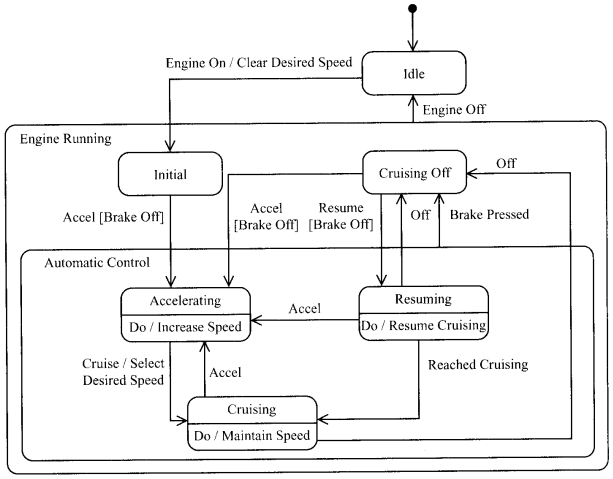
2. Consider the effect of all alternative external events which could lead to additional transitions being produced:
   - ACCEL event: can occur in the *Resuming*, *Cruising* or *Cruising Off* states → transition to *Accelerating*
   - BRAKE pressed event: can occur while in the *Accelerating*, *Cruising* or *Resuming* states → transition to *Cruising Off*
   - OFF event: can occur while in *Resuming* or *Cruising* states → transition to *Cruising Off*
   - ENGINE off event: occurs in any state → transition to *Idle* state.



[Gomaa, 2000]

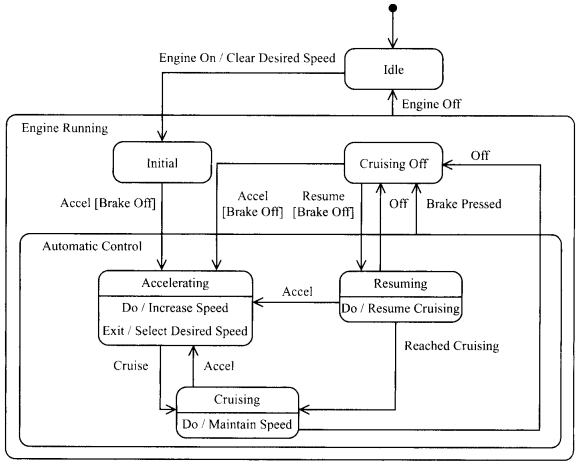
Mapping to Hierarchical or Orthogonal Statecharts

Look for natural aggregations of states to create superstates to simplify the flat statechart, e.g. consider *Accelerating*, *Cruising*, and *Resuming* as substates of an *Automatic Control* superstate:



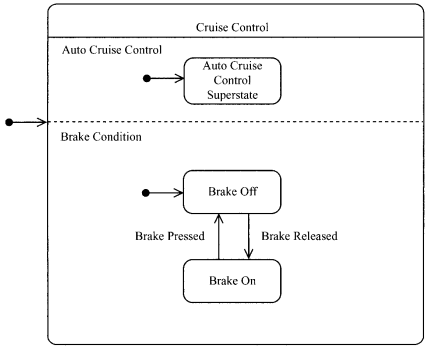
[Gomaa, 2000]

The *Brake Pressed* transition from the *Accelerating* state has a *Select Desired Speed* action → shift to an exit action from *Accelerating*:



[Gomaa, 2000]

Other aspects of the *Cruise Control* object can also be modelled, e.g. the *Brake Condition* can be maintained by an additional orthogonal statechart to the Auto Cruise Control statechart already described:



[Gomaa, 2000]

Note that in the COMET methodology, orthogonal statecharts are used to show concurrently held states, and not concurrent execution. Separate statecharts should be used to describe concurrently executing objects.

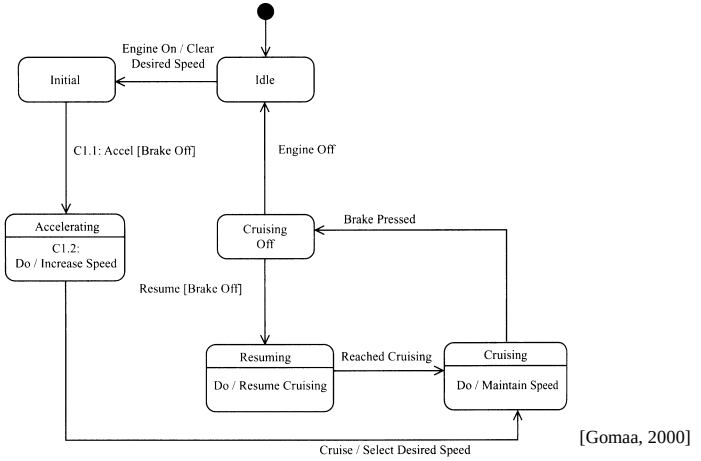
Example: State-dependent dynamic analysis - Cruise Control system

The aim of the analysis is to confirm the operation of the statechart developed from the use case scenarios and to develop all the object interactions from external inputs through to outputs in the form of a collaboration diagram.

The use case description will reveal the need for the state-dependent *Cruise Control* object to encapsulate the statechart and the various device interface objects: *Cruise Control Lever Interface, Engine Interface, Brake Interface*. Since final output of the use case is to adjust the throttle position a *Throttle Interface* object is also needed.

The sequence of events through each scenario of the use case is traced via an event sequence analysis, matching collaboration diagram messages sequence numbers to statechart transitions.

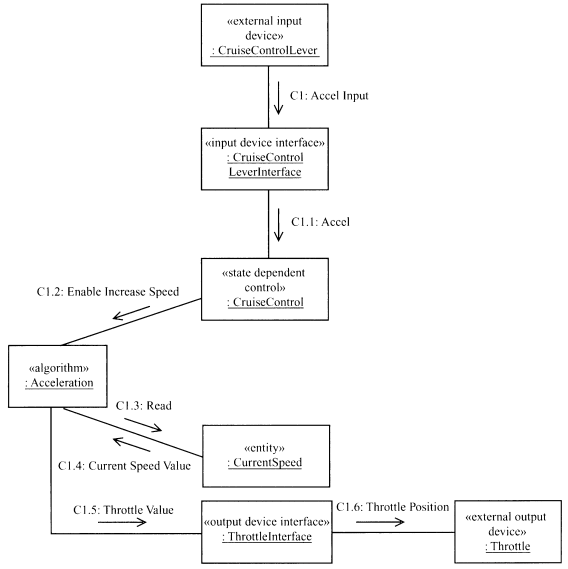
Consider the first external event ACCEL and the statechart response to it:



[Gomaa, 2000]

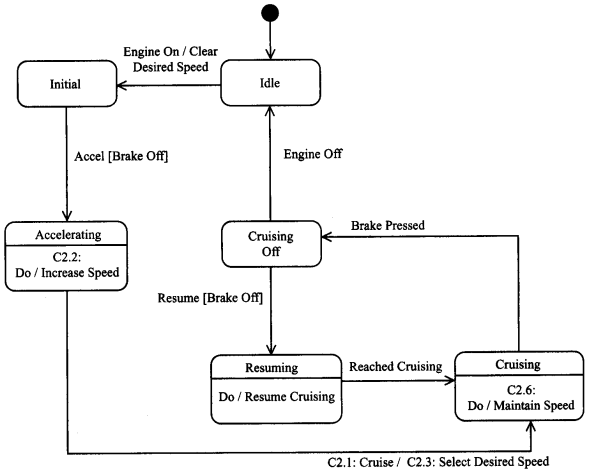
The corresponding event sequence on the collaboration diagram:

Increase Speed is the primary activity - it enables the *Acceleration* object that determines the throttle value to gradually increase the vehicle speed.



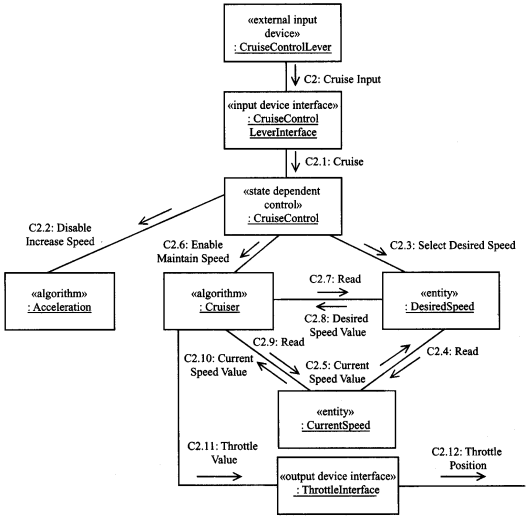
[Gomaa, 2000]

The next external event is CRUISE when the cruise control lever is released:



[Gomaa, 2000]

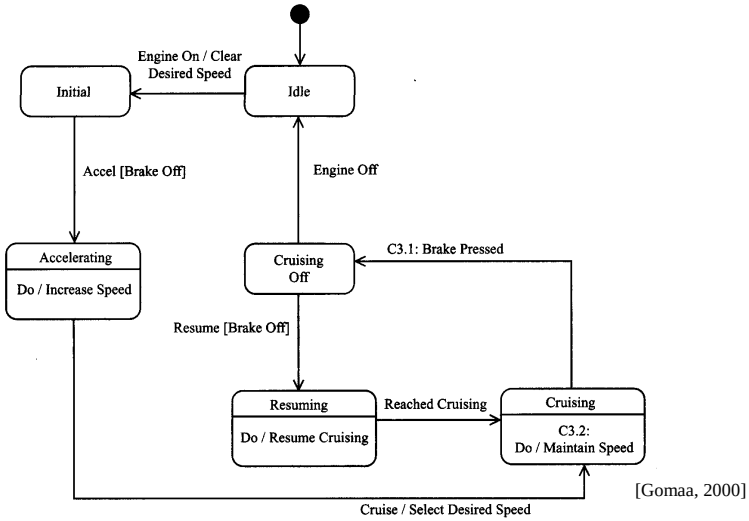
The corresponding event sequence on the collaboration diagram:



[Gomaa, 2000]

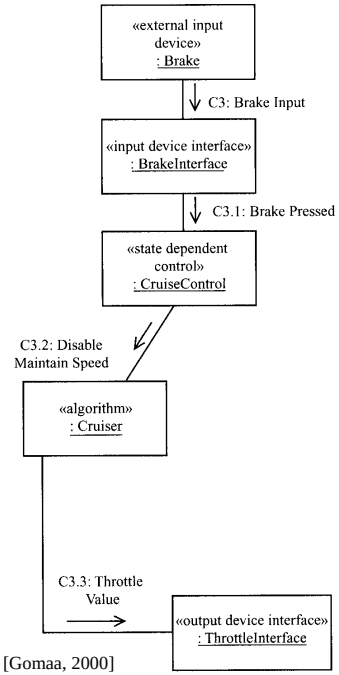
Maintain Speed is the primary activity - it enables the *Cruiser* object that determines the throttle value to maintain the vehicle speed.

The next external event is BRAKE when the brake is pressed:



[Gomaa, 2000]

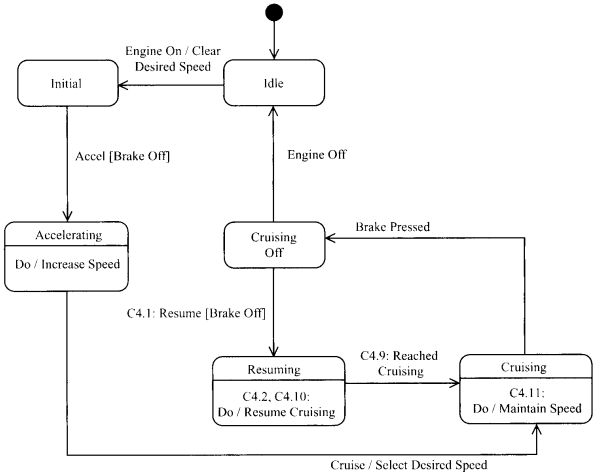
The corresponding event sequence on the collaboration diagram:



[Gomaa, 2000]

Maintain Speed is disabled and the *Throttle Value* sent indicates that manual control is resumed.

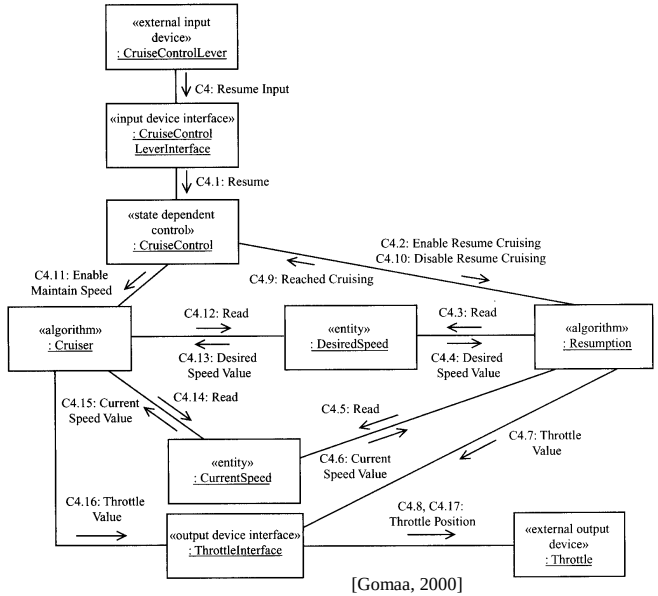
The next external event is RESUMING when the cruise control lever is set to *Resume*:



[Gomaa, 2000]

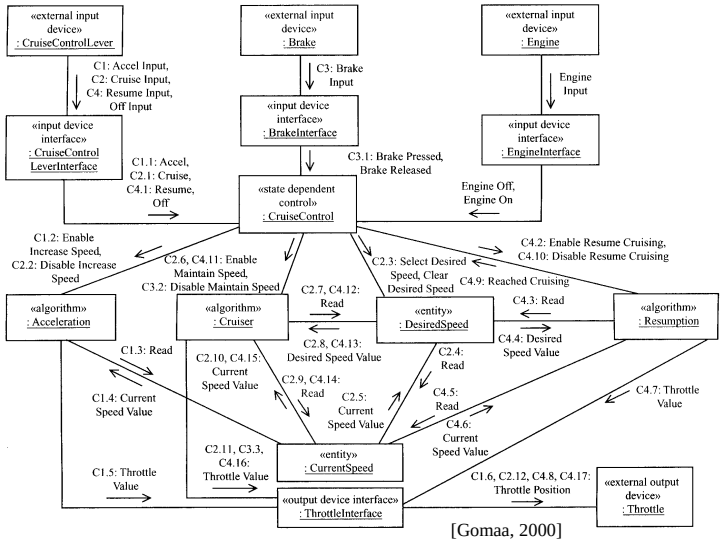
The corresponding event sequence on the collaboration diagram:

Resume Cruising is the primary activity - it enables the *Resumption* object that determines the throttle value to resume the vehicle cruise speed before handing over to the *Cruiser* object to maintain the vehicle speed.



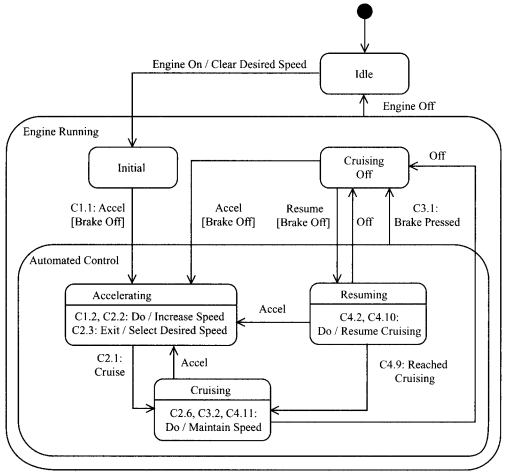
[Gomaa, 2000]

Based on all the alternative events, the final collaboration diagram for this use case scenario can be created:



[Gomaa, 2000]

And a hierarchical statechart can be produced showing all the collaboration diagram labelled activities and transitions:



[Gomaa, 2000]

These diagrams form the basis for the next step in the COMET methodology which is the development of the *software architecture* design.