Statemate Course

Kai Baukus

Statemate/SDL

W.-P. de Roever	D. Hogrefe
K. Baukus	H. Neukirchen
CAU Kiel	MU Lübeck

Activity-Charts and Statecharts

We discuss the actions used by a statechart to control activities, and the events and conditions used by it to sense their status.

[HP98] Modeling Reactive Systems with Statecharts: The STATEMATE Approach, D. Harel, M, Politi. McGraw-Hill, 1998.

SCL programs

Skeleton

```
PROGRAM  template;

CONSTANT
 INTEGER step_counter := 0;
VARIABLE
 INTEGER I := 45;
 GLOBAL FILE f1;

INIT
END INIT;

SET BREAKPOINT [STEP] DO
END BREAKPOINT;

BEGIN
END;
END.
```

Declaration

```
-- CONSTANT
--%% To declare constants remove the comment sign in the previous line
--%% and declare their types as in the example in the next several line
--%%    INTEGER step_counter := 0;

-- VARIABLE
--%% In this section you declare variables
--%% and their types as in the example in the next several lines.
--%%    INTEGER I := 45;
--%%    GLOBAL FILE f1;
```

Init

```
-- INIT
--%% In this section you place SCL statements to be executed once,
--%% each time the SCP is invoked.
-- END INIT;



INIT
time_to_change:=0;
open(fout,'/tmp/klimatest.log', output);
write(fout,'Juhu','\n')
END INIT;
```

Breakpoints

```
-- SET BREAKPOINT [STEP] DO
--%% In this section you put statements to be executed conditionally,
--%% at each simulation step.  Duplicate this structure for each
--%% breakpoint definition.
--%% The condition STEP may be replaced by other condition/event expres
-- END BREAKPOINT;

set breakpoint neue_temperatur=> every 1 do
  if (time_to_change <= cur_clock) then
    temp:=rand_iuniform(-20,40);
    write('Temperature changed to ',temp,' at ',cur_clock,'\n');
    time_to_change:=rand_iuniform(20,120)+cur_clock;
    write('Next change at ',time_to_change,'\n');
  end if;
end breakpoint;
```

Body

```
-- BEGIN
--%% This is the Main Section, in which you place any SCL statements,
--%% including Go statements.
--%% These statements are executed sequentially.
-- END;

BEGIN
write('Simulation started at: ',cur_clock,'\n');
while true loop
  go step;
end loop;
```

Connections between Activity-Charts and Statecharts

Dynamics in the Functional Decomposition

The activities participating in the functional decomposition are not necessarily always active. In most kinds of systems many of the activities have limited periods in which they are active.

Examples

Some examples with different dynamics:

- **Procedures and functions** in software programs start when they are "called" and stop upon completion.
- In **multi-tasking or multi-processing systems**, tasks are invoked, do their job, and then are "killed" or "kill" themselves.
- Tasks with **lower priority** maybe interrupted or delayed when a mission of higher priority arrives.
- **Interactive user interface** is specified by "callback functions".

EWS Example

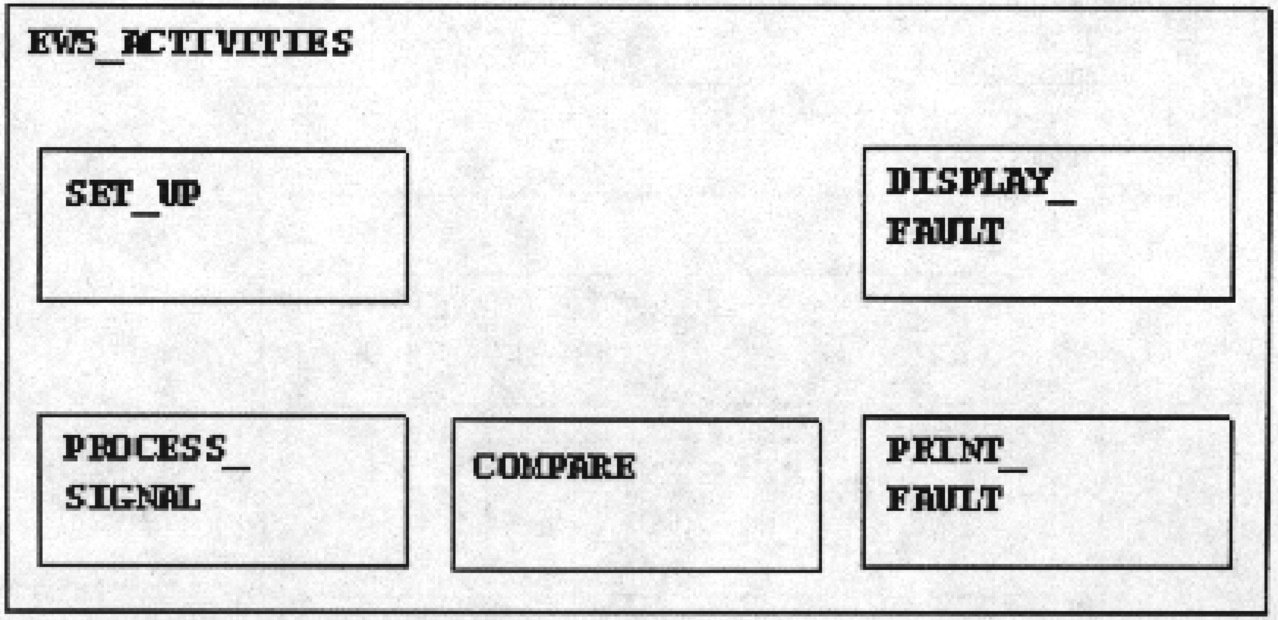


Figure 2.2. First level decomposition of an activity

EWS Example (cont'd)

Dynamic and timing issues related to the activities in the EWS example:

SET_UP: activated by an explicit request of the operator, terminates on its own.

COMPARE: starts with the EXECUTE command and stops with the OUT_OF_RANGE event or the RESET command.

PROCESS_SIGNAL: active when the COMPARE activity is active.

DISPLAY_FAULT: starts with the OUT_OF_RANGE event and is stopped by the operator or after a predefined time period.

PRINT_FAULT: starts when the time period is passed and terminates on its own.

Specifying the Dynamics

- Obviously, merely listing the activities and their connections, as is done in the functional view, is not sufficient. We have to specify the dynamics of controlling these activities, including the starting and stopping of the subactivities of a non-basic activity.
- The order in which the functional and behavioral views and their connections are developed depends on the nature of the system and on the specification methodology.

Methodology

- One can start by carrying out a functional decomposition in activity-charts, and then add the timing and other dynamic information in statecharts to capture behavior.
- In contrast, it is possible to start by using statecharts to describe the system’s modes of operation and/or a collection of use-cases, and then construct an activity-chart from the activities performed in these modes or scenarios.

Dynamics of Activities

In order to capture the dynamic behavior of non-basic activities, our models employ control activities that are associated with statecharts.

Statecharts in the functional view

When a non-basic activity that contains a control activity starts its execution, the statechart associated with that control activity becomes active, i.e., the system enters the top level state of this statechart.

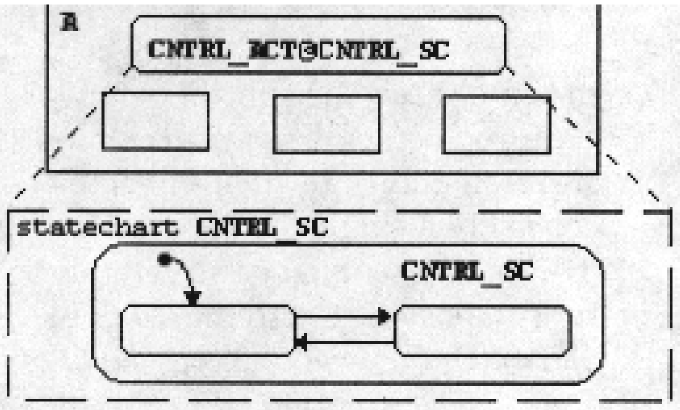


Figure 7.1. Associating a statechart with a control activity

Statemate MAGNUM

Control Activities

An activity with a reactive behavior pattern can be described by a statechart even though it is not further decomposed, so that it has no subactivities to control:

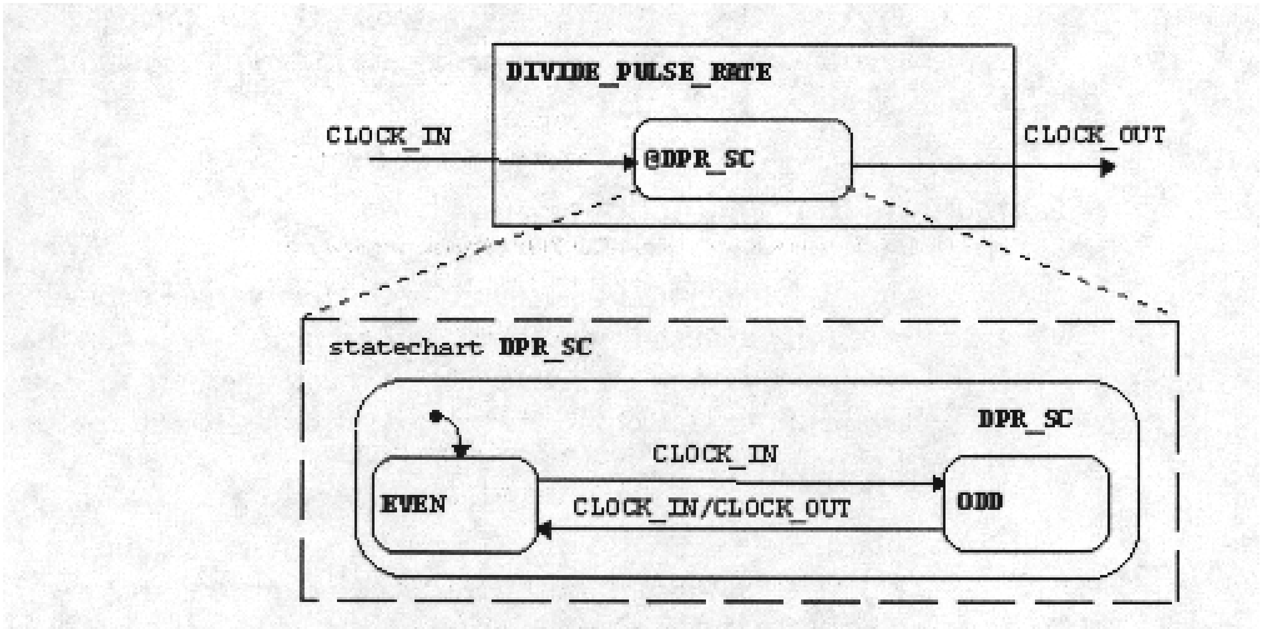


Figure 7.2. A statechart describing a simple activity

Control Activities (cont'd)

In some cases, the control behavior of an activity can be captured by static reactions alone. Then, the controlling statechart will consist of a single top-level state with the static reactions given in its Data Dictionary entry.

Note: While the controlling statechart may consume and produce external (control and data) information, its interface does not appear in the statechart itself.

Termination Type of an Activity

We distinguish between activities that have **self-termination** and those that have **controlled-termination.**

If a self-terminating activity has a control activity, then the corresponding statechart must contain a **termination connector.**

Termination Connector

It is considered a final state; in particular it has no exits:

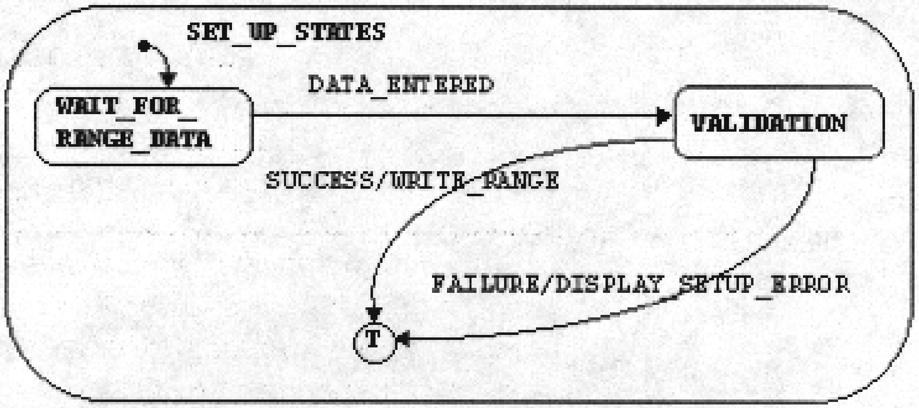


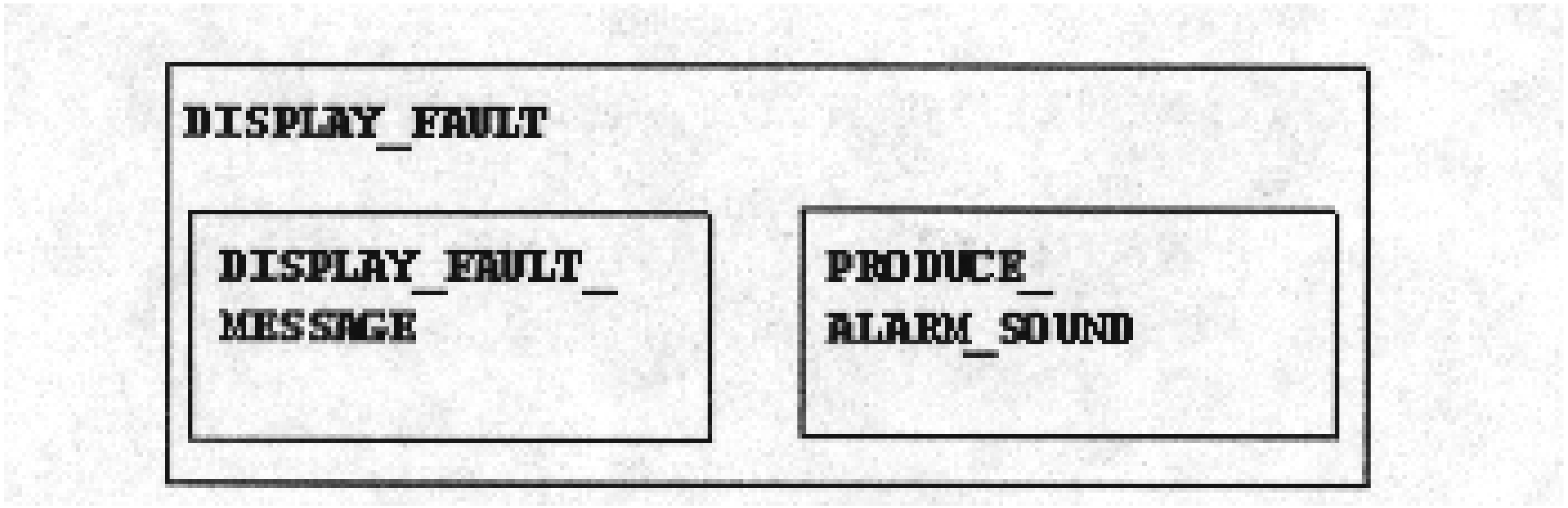
Figure 7.3. Termination connector in **SET-UP**'s statechart

Upon entering this connector, the statechart "stops", its parent activity becomes deactivated, and the event STOPPED(A) occurs.

When a non-basic activity stops, all its subactivities stop immediately too.

Perpetual Activities

Sometimes there is no control activity needed:



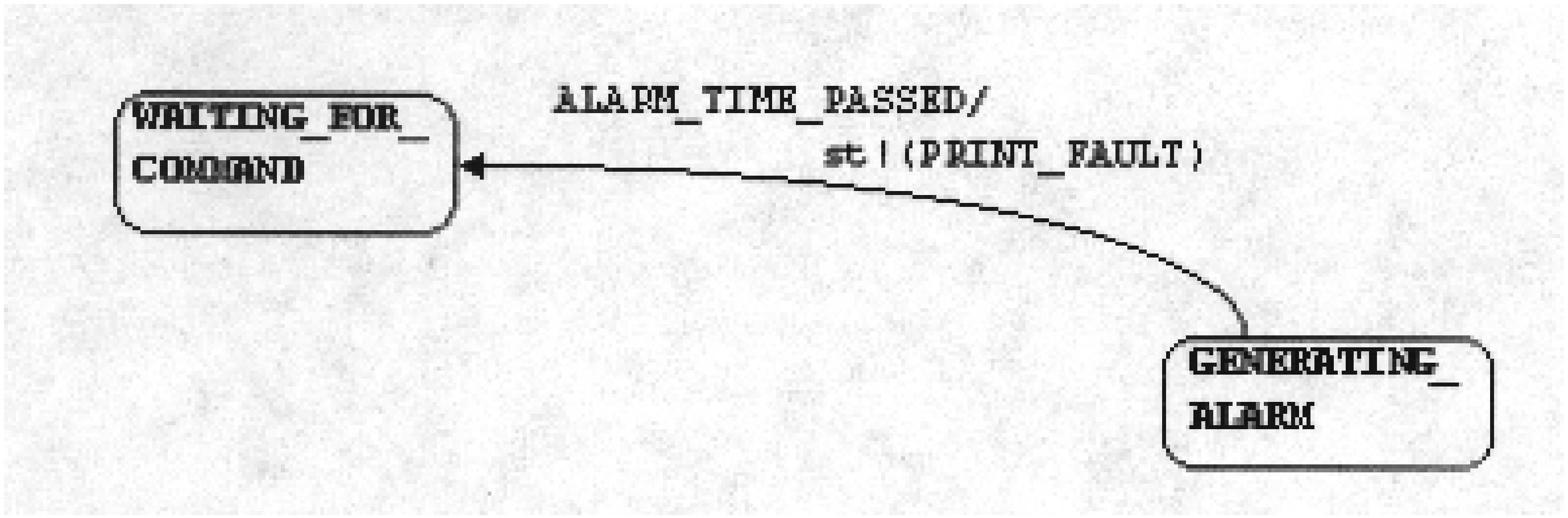
All the subactivities start when the parent activity starts, and they all stop when it stops.

Controlling the activities

How does the controlling statechart affect and sense the status of its sibling activities?

Starting and Stopping Activities

The main mechanism that statecharts use to control activities is the ability to start and stop them explicitly:



Examples

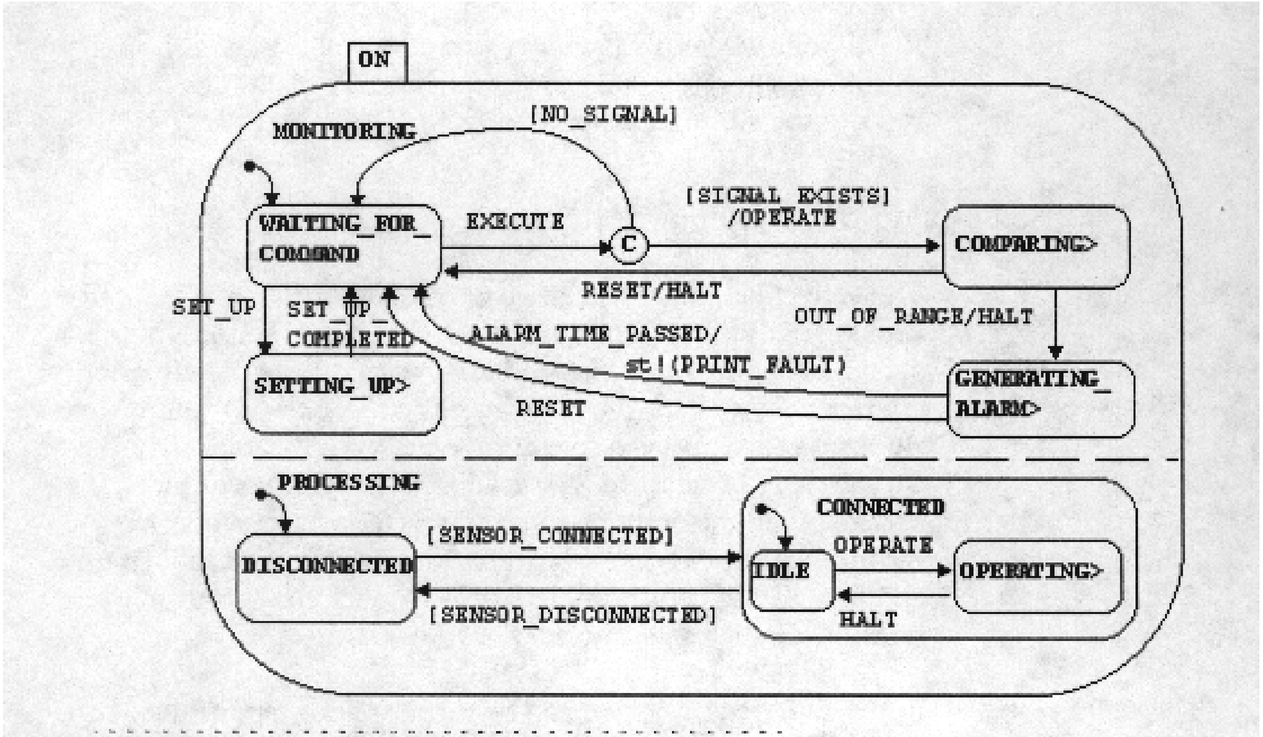


Figure 7.6. States marked as having entering and exiting reactions

The control activity can control only its sibling activities. Therefore, all actions that appear in its statchart may refer to the sibling activities only.

Sensing the status of Activities

The statchart that describes a control activity is not limited to causing activities to start and stop. It can also sense whether such happenings have indeed taken place. Specifically, the control activity can sense the events STARTED(A) and STOPPED(A), and the condition ACTIVE(A).

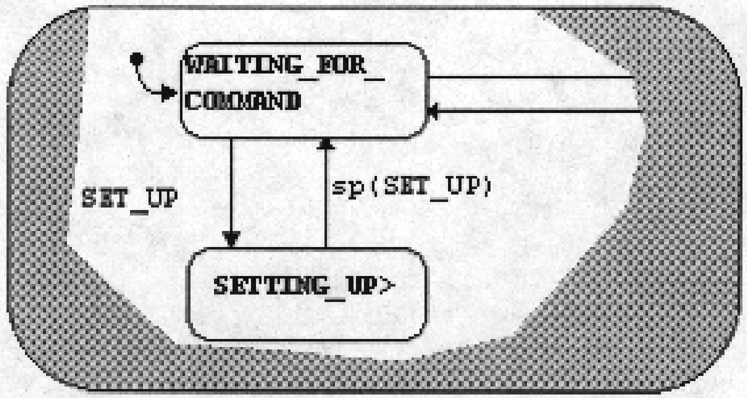


Figure 7.7. An event signifying termination of an activity

The events and conditions in the describing statechart are allowed to refer only to the sibling activities.

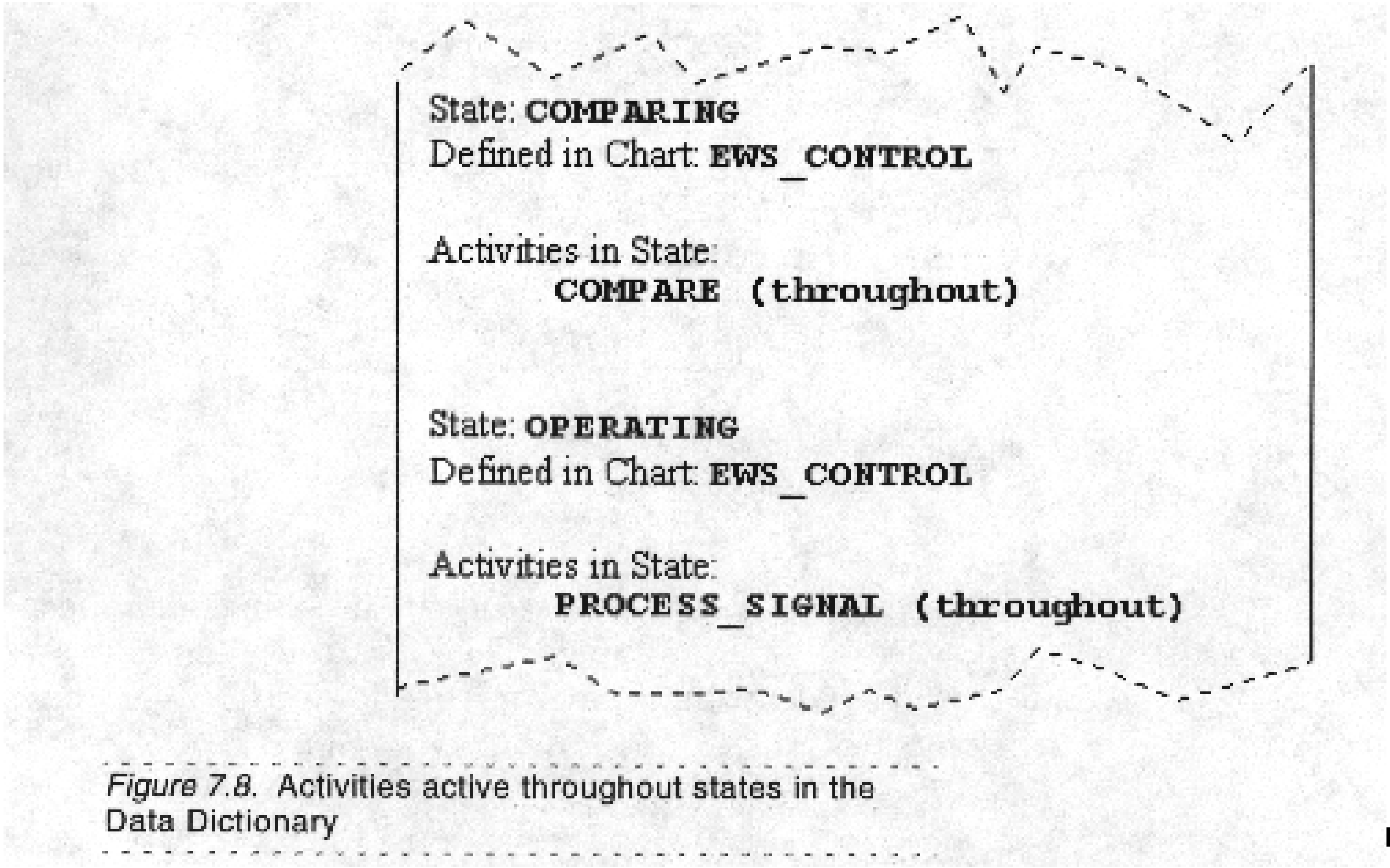
Activities in States

Often, we wish an activity A to start when a certain state S is entered, and to stop when S is exited.

This can be specified by associating the action ST!(A) with the entering event NS and SP!(A) with the exiting event XS in the Data Dictionary.

Activities Throughout and Within States

Another more compact way is to specify that A is **active throughout** S:



State: COMPARING
Defined in Chart: EWS_CONTROL

Activities in State:
COMPARE (throughout)

State: OPERATING
Defined in Chart: EWS_CONTROL

Activities in State:
PROCESS_SIGNAL (throughout)

Figure 7.8. Activities active throughout states in the Data Dictionary

Suspending and Resuming Activities

In addition to being able to start and stop activities, control activities can cause an activity to “freeze”, or **suspend**, its activation, and to later **resume** from where it stopped.
The relevant actions are SUSPEND(A) and RESUME(A).
Associated with these actions is the condition HANGING(A).

Suspension

Suspension may be used, for example, when we want to interrupt the progress of an activity in favor of another activity with higher priority:

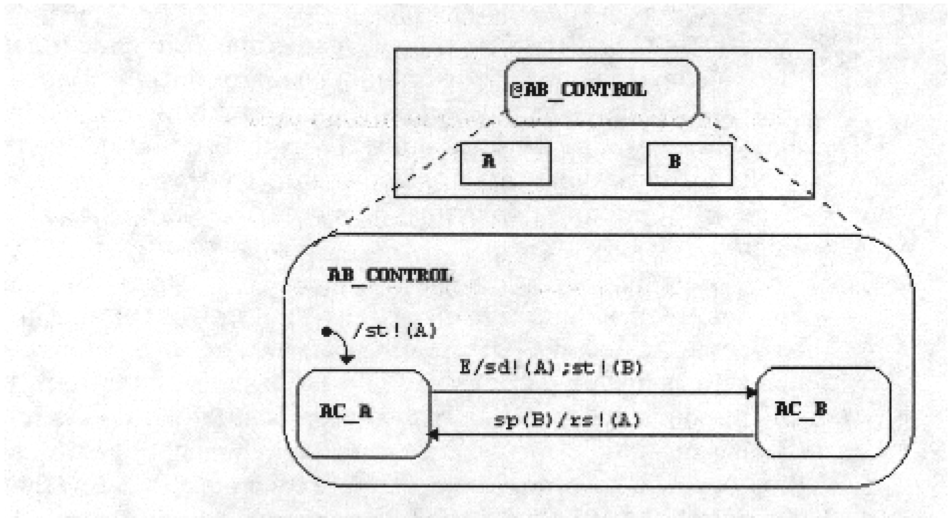


Figure 7.10. Suspending and resuming activities

The event E causes A to be suspended, while the preferred activity B is carried out to completion, at which time A is resumed.

Comparison with History Entrance

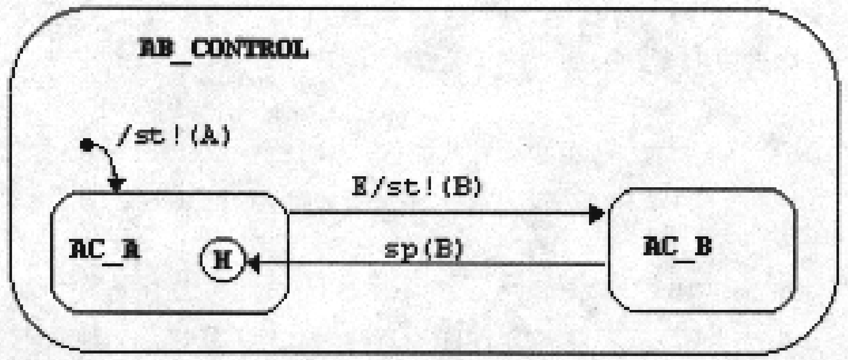


Figure 7.11. History entrance vs. resume activity

When A is active throughout AC_A the action is started again.

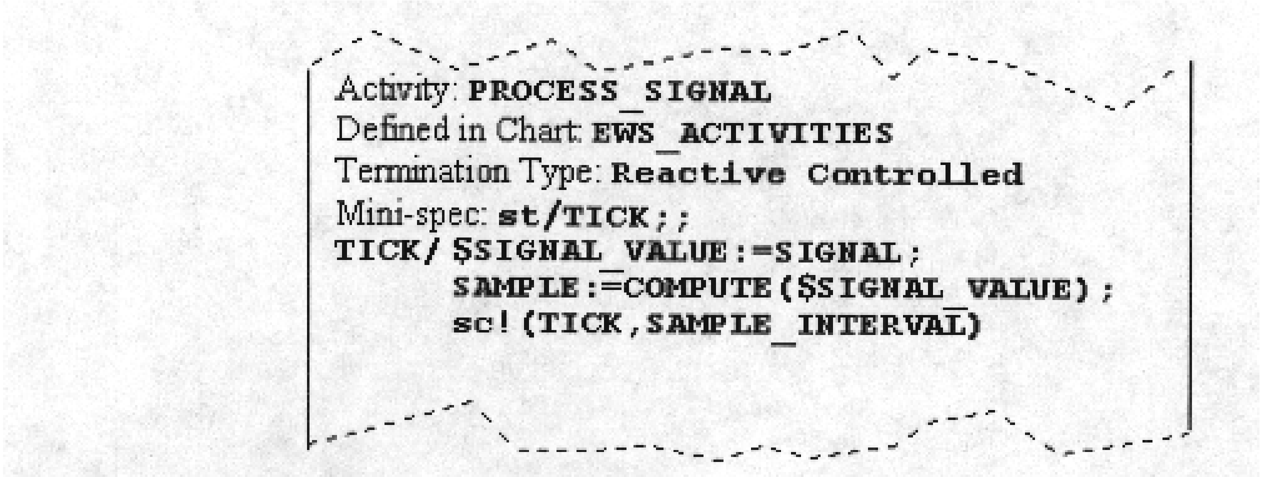
On the other hand, not stopping A would allow A to react on events while B is active.

Specifying Behavior of Basic Activities

When carrying out functional decomposition, the lower building blocks of the description are the basic activities, those that require no further breakup.
Basic activities may have additional textual descriptions in the Data Dictionary and are marked by a “$>$”.

Reactive Mini-Specs

In some cases the behavior of a basic activity can be described by a collection of reactions, consisting of triggers and their implied actions.
A reactive mini-spec is a list of reactions of the form TRIGGER/ACTION, separated by a double semi-colon (;;).



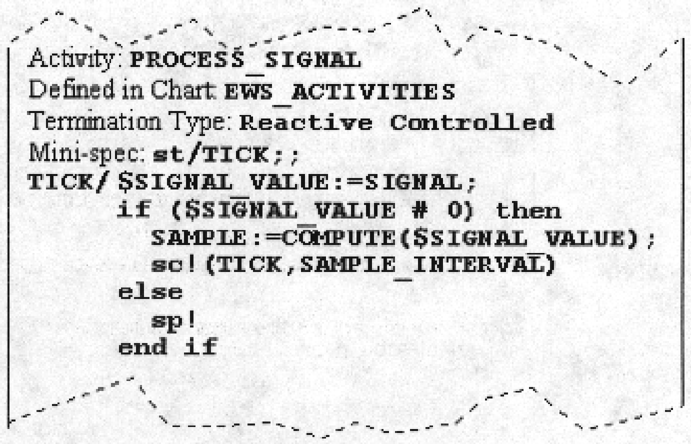
Activity: **PROCESS_SIGNAL**
Defined in Chart: **EWS_ACTIVITIES**
Termination Type: **Reactive Controlled**
Mini-spec: **st/TICK;;**

```
TICK/ $SIGNAL_VALUE:=SIGNAL;
      SAMPLE:=COMPUTE($SIGNAL_VALUE);
      sc!(TICK,SAMPLE_INTERVAL)
```

Figure 7.12. A reactive mini-spec in the Data Dictionary

Reactive Mini-Specs (cont'd)

A reactive mini-spec can be attached to both self-terminating or controlled-terminating activities.



```
Activity: PROCESS_SIGNAL
Defined in Chart: EWS_ACTIVITIES
Termination Type: Reactive Controlled
Mini-spec: st/TICK;;
TICK/ $SIGNAL_VALUE:=SIGNAL;
      if ($SIGNAL_VALUE # 0) then
        SAMPLE:=COMPUTE($SIGNAL_VALUE);
        sc!(TICK,SAMPLE_INTERVAL)
      else
        sp!
      end if
```

Figure 7.13. The **stop** action in a mini-spec

It is important to remember that states and activities cannot be referred to in the mini-spec. All the activities and states of the model are beyond the scope of an individual mini-spec.

Procedure-Like Mini-Specs

Often an activity can be described as a sequence of actions, possibly with conditional branching and iterations. Such activities are called **procedure-like.** They are active for a single step only. Therefore, such activities are always self-terminating.

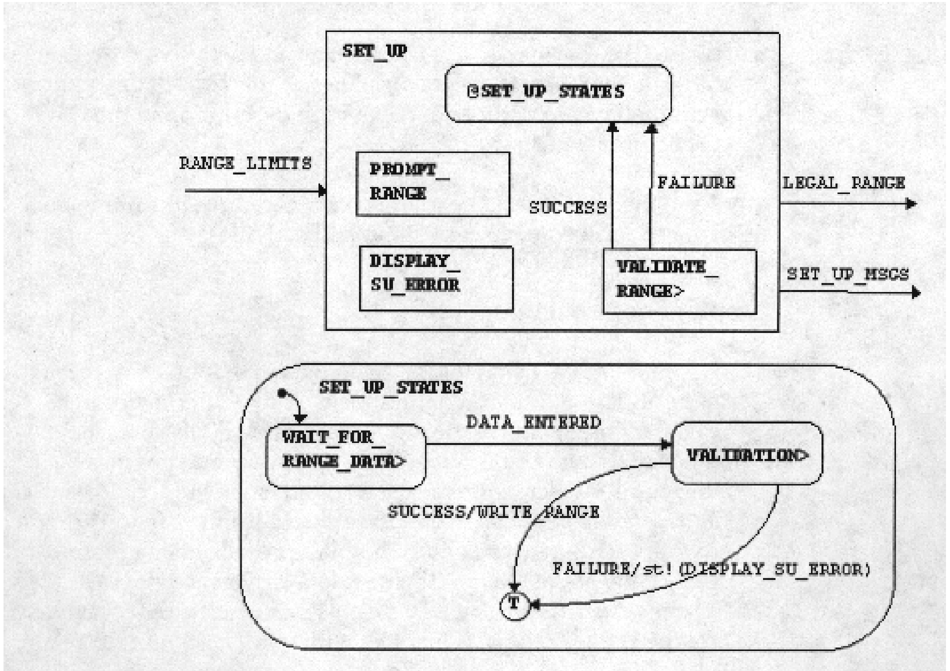
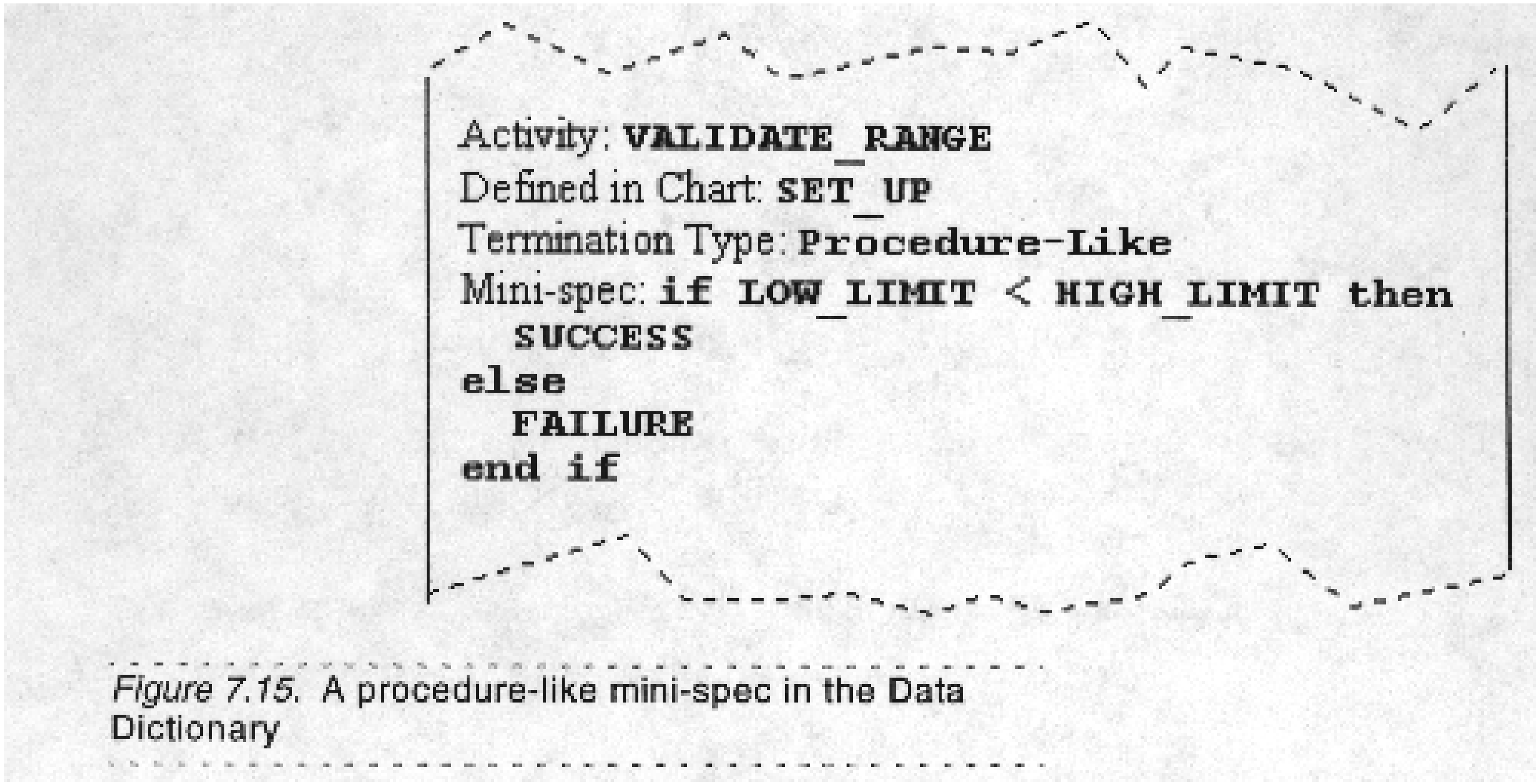


Figure 7.14. **SET-UP** activity and controlling statechart

Corresponding Mini-Spec



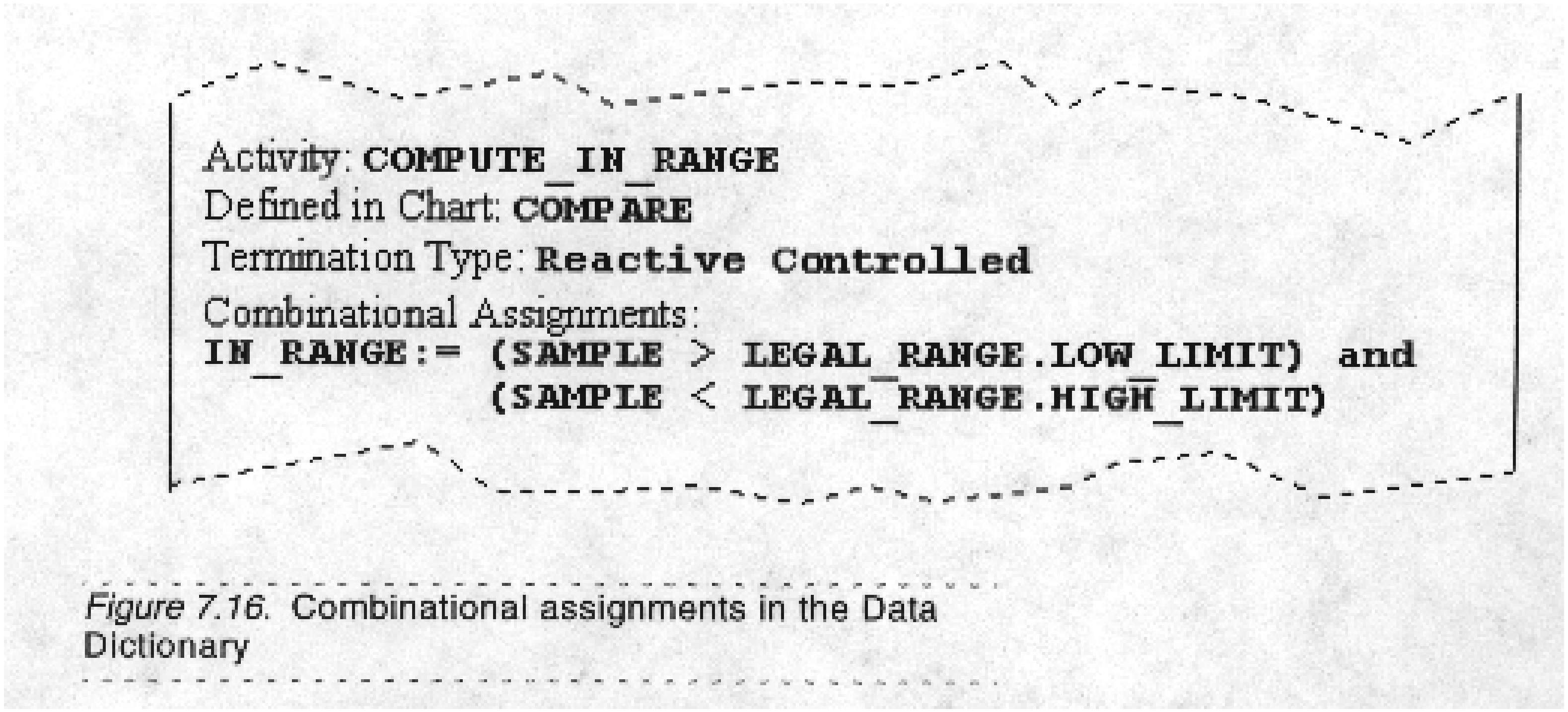
Activity: **VALIDATE_RANGE**
Defined in Chart: **SET_UP**
Termination Type: **Procedure-Like**
Mini-spec:

```
if LOW_LIMIT < HIGH_LIMIT then
  SUCCESS
else
  FAILURE
end if
```

Figure 7.15. A procedure-like mini-spec in the Data Dictionary

Combinational Assignments

Another typical behavior for an activity is that of a **data-driven** pattern. The activity is continuously ready to perform some calculations whenever the input changes its value.



Activity: **COMPUTE_IN_RANGE**
Defined in Chart: **COMPARE**
Termination Type: **Reactive Controlled**
Combinational Assignments:

```
IN_RANGE:= (SAMPLE > LEGAL_RANGE.LOW_LIMIT) and
           (SAMPLE < LEGAL_RANGE.HIGH_LIMIT)
```

Figure 7.16. Combinational assignments in the Data Dictionary

Communication between Activities

Specifying the communication between activities consists of the what and the when, just like for other parts of the specification.
The what is described by the flow-lines in the activity-charts and relevant parts of the Data Dictionary. The when is to be specified by the behavioral parts of the model, i.e., the statecharts and mini-specs.

Communication and Synchronization Issues

Functional components in systems communicate between themselves in order to pass along information and to help synchronize their processing. A number of attributes characterize the various communication mechanisms.

Attributes

Communication can be

- **instantaneous** , meaning that it is lost when not consumed immediately, or

  persistent , meaning that it stays around until it gets consumed.
- **synchronous** , i.e., the sender waits for an acknowledgment, or

  asynchronous , i.e., there is no waiting on the part of the sender
- **directly addressed** , i.e., the target is specified, or sent by

  broadcasting

Controlling the Flow of Information

In the following figure x is specified to flow between activities A and B:

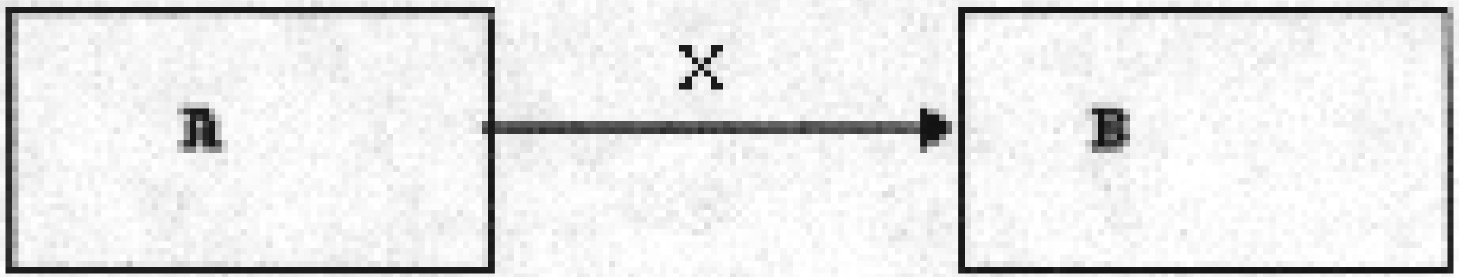


Figure 8.1. An information element flowing between activities

Flow of Information

If x is an event we may have the following situation:

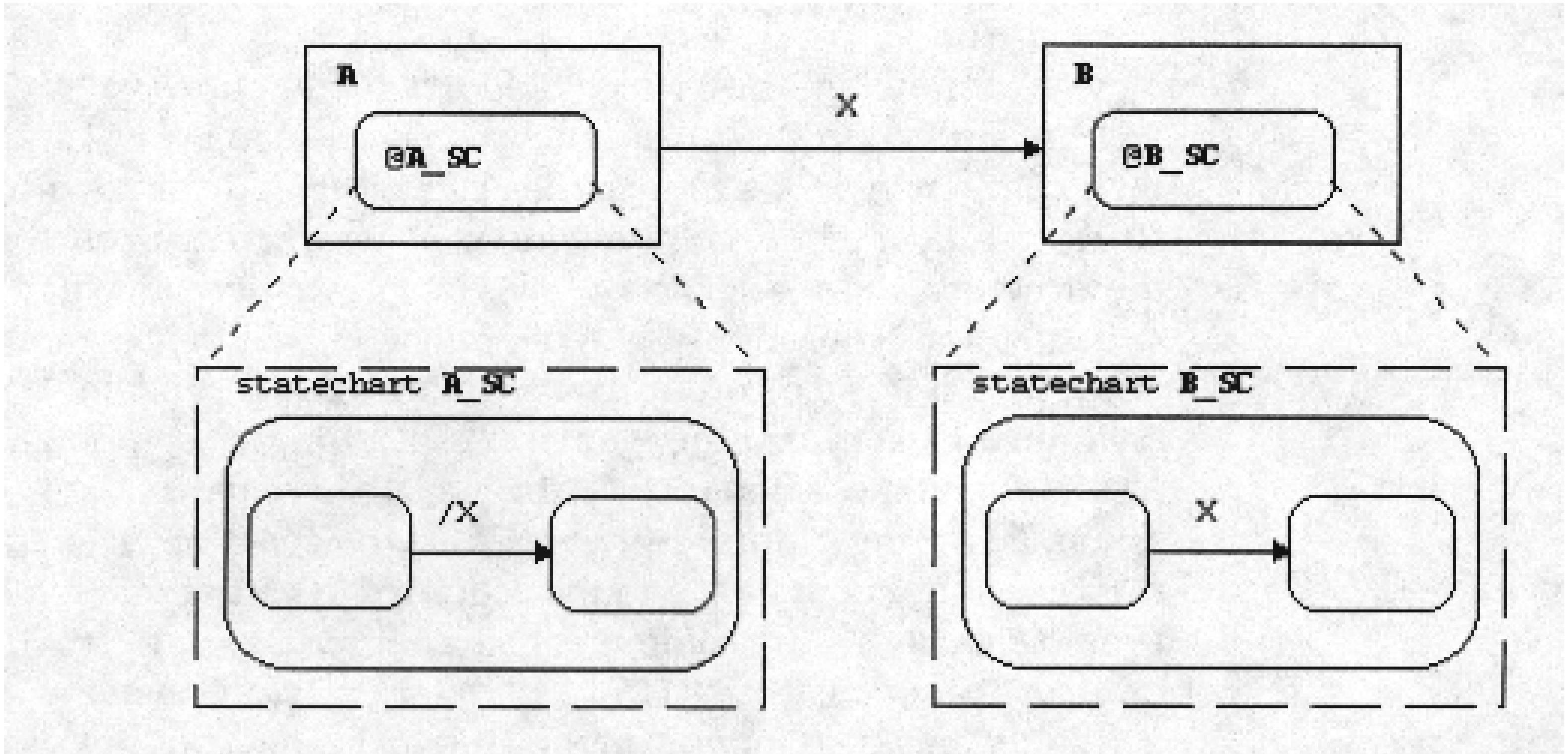


Figure 8.2. Producing and consuming an event

If x is a condition or data-item modified by A, B could sense the value or the change of the value (x, TR(x), WR(x)).

Examples of Communication Control

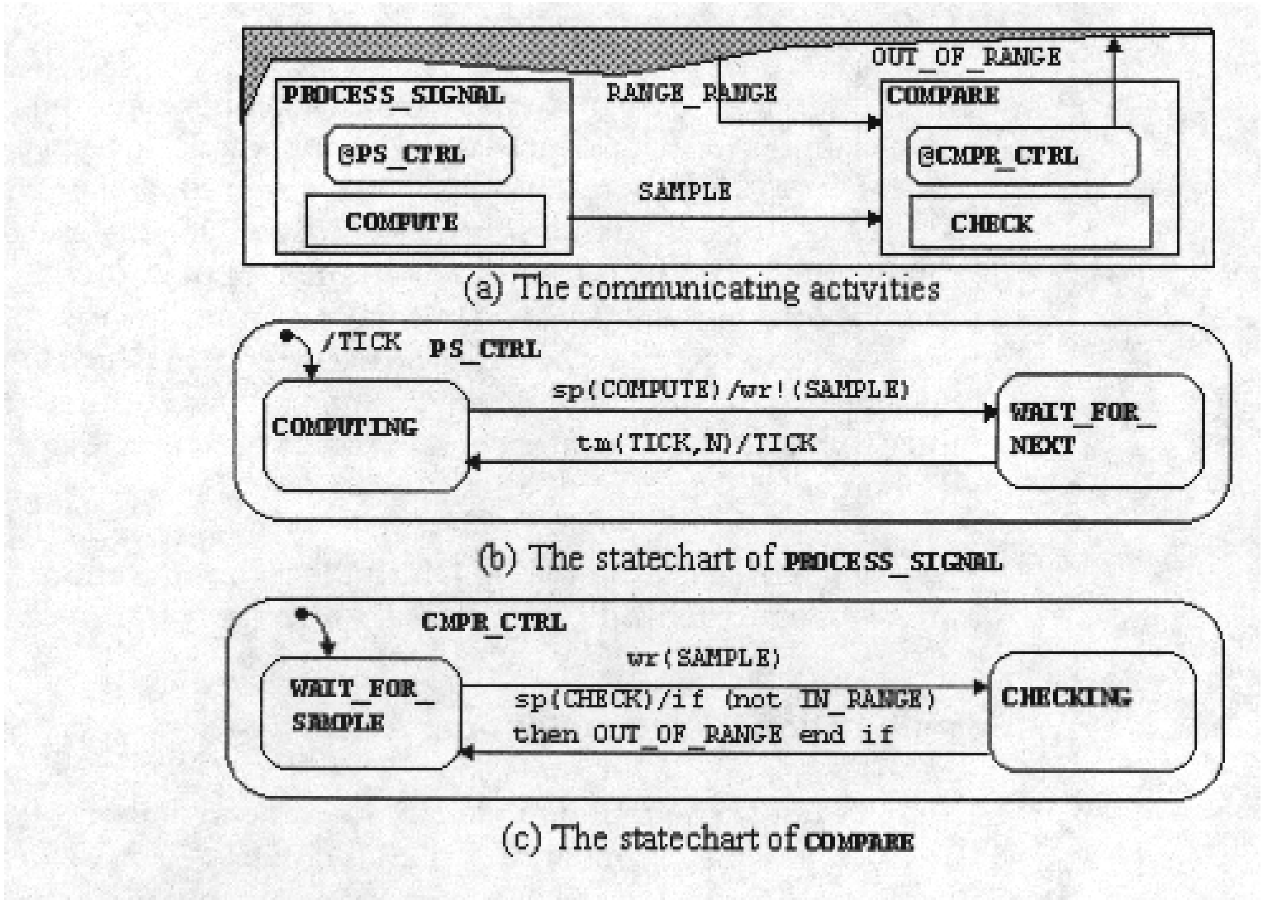


Figure 8.4. Communication between periodic activities

Message Passing

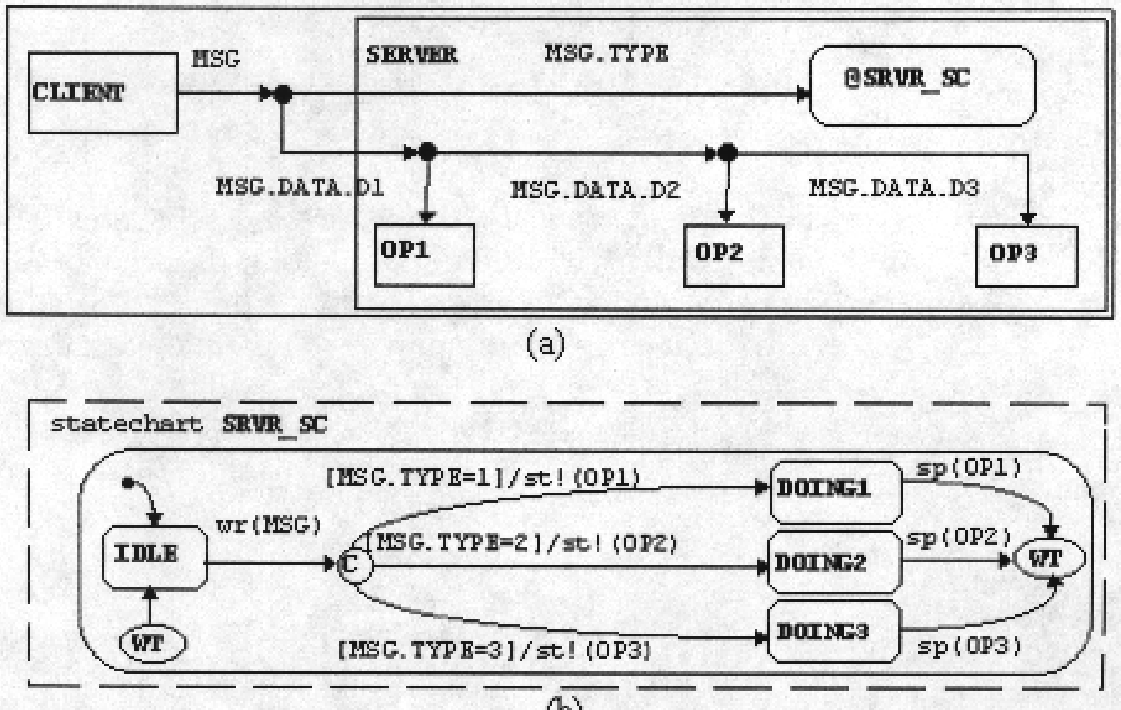


Figure 8.5. Server responding to three service requests

Activities Communicating Through Queues

Queuing facilities for messages are virtually indispensable in modeling multi-processing environments, and especially multiple client-server systems.
We want to have:

- ability to sent unlimited number of messages to the same address, while the receiver is not always in a position to accept them,
- no message is consumed before one that was sent earlier,
- possibility for concurrently active components to write messages to the same address at the same moment
- possibility for concurrently active components to read different messages to the same address at the same moment

Queues and their operations

A queue is an ordered, unlimited collection of data-items, all of the same data type. The queue is usually shared among several activities, which can employ special actions to add elements to the queue and read and remove elements from it.

- **q_put(Q,D)** add the value of expression D to the queue
- **q_urgent_put(Q,D)** add the value of expression D to the head of the queue
- **q_get(Q,D,S)** extract the element at the head of Q and place it in D
- **q_peek(Q,D,S)** same as above without removing the element from Q
- **q_flush(Q)** clears Q totally

Combination with Data Stores

The following figure illustrates the order in which operations on a queue are performed during a step:

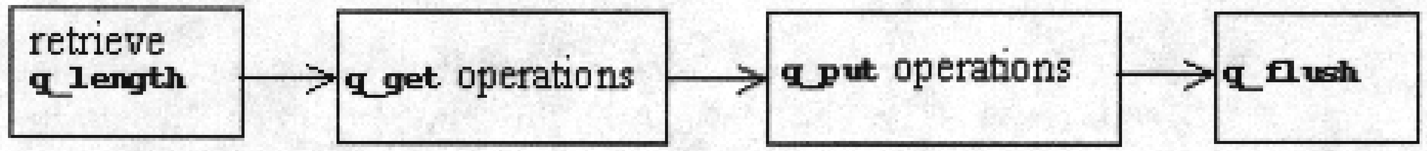


Figure 8.6. Operations on a queue during a step

Queues can be associated with data stores just like data-items of other types can.

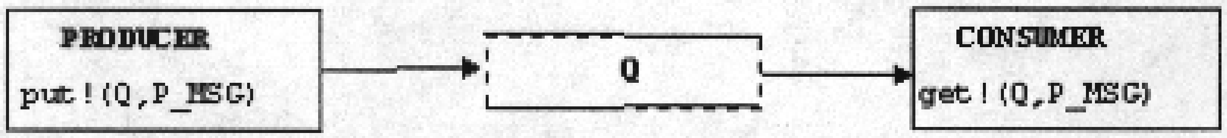


Figure 8.7. A queue associated with a data-store

Example

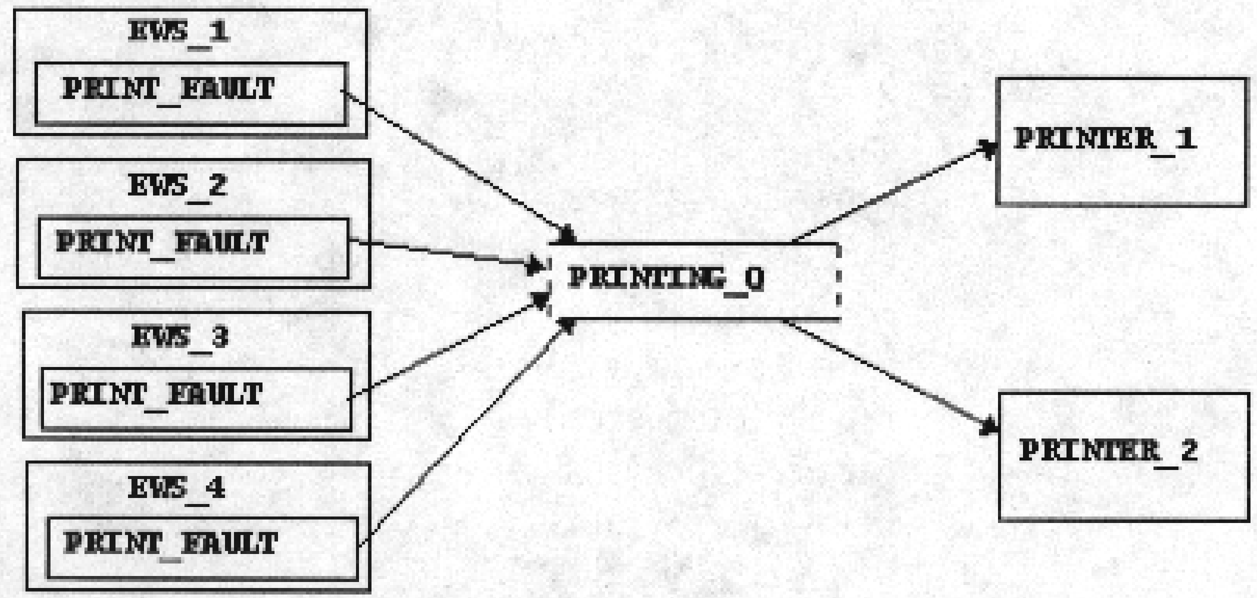
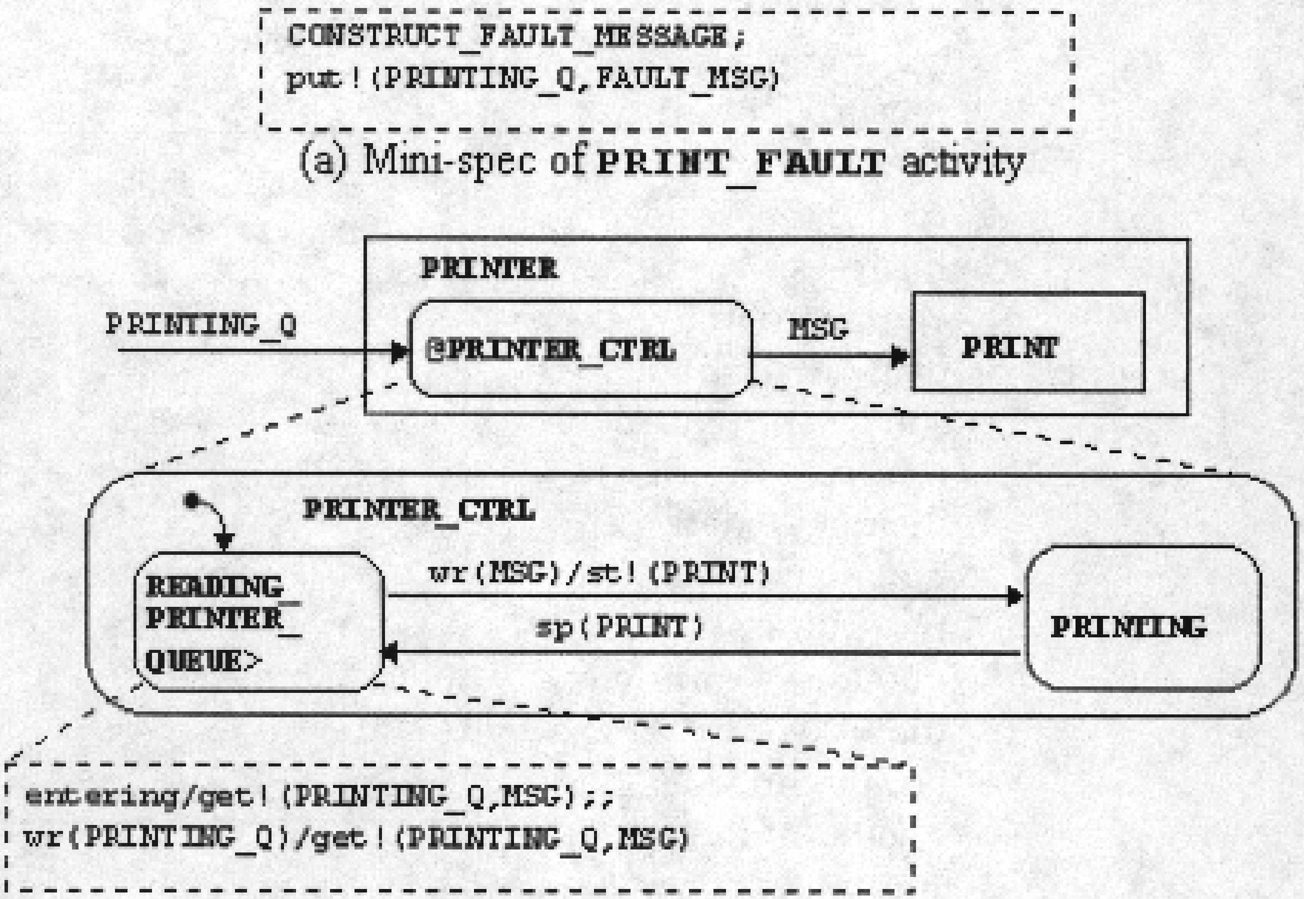


Figure 8.8. Multiple clients served by multiple servers via a queue

Example



(b) The description of the PRINTER

Figure 8.9. Writing and reading messages from a queue

Conditions and Events Related to States

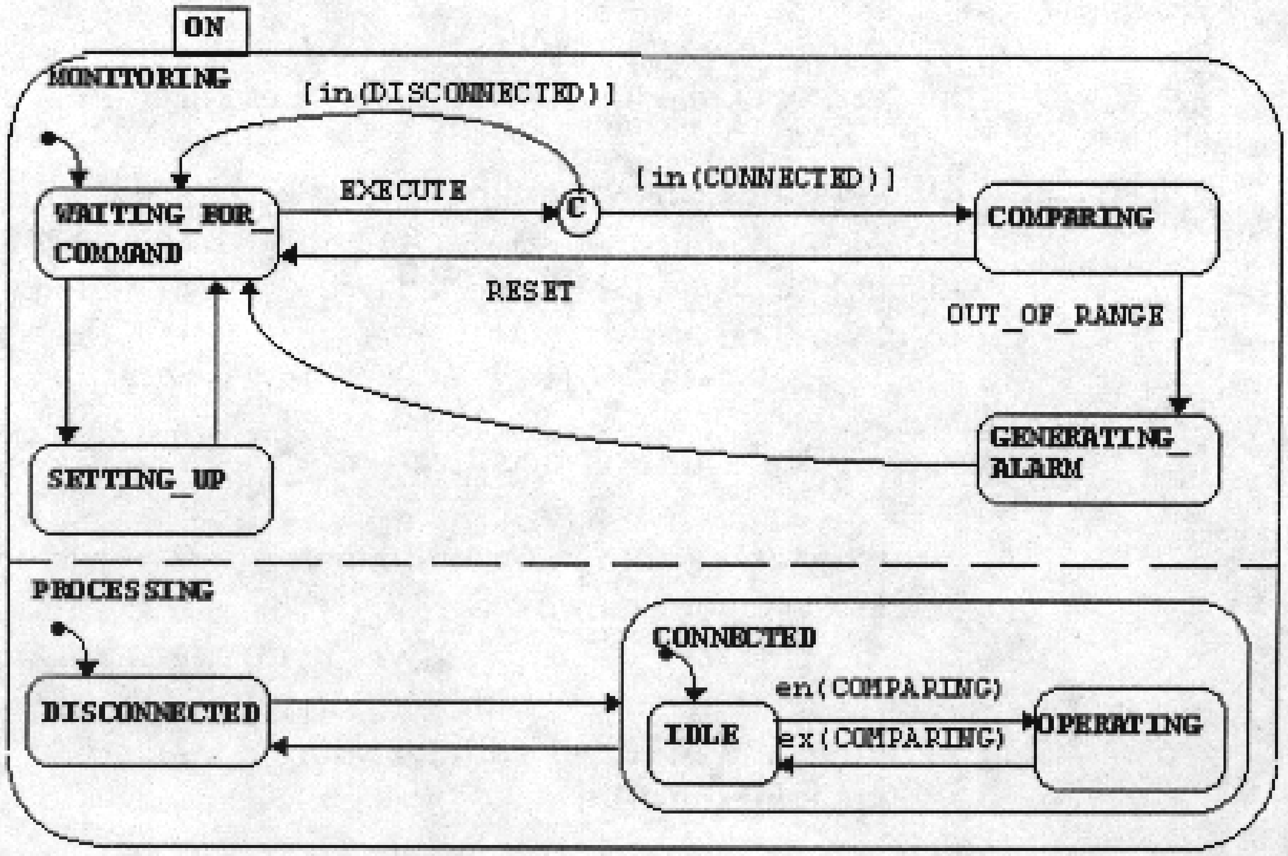


Figure 4.15. Conditions and events related to states

Condition Connector

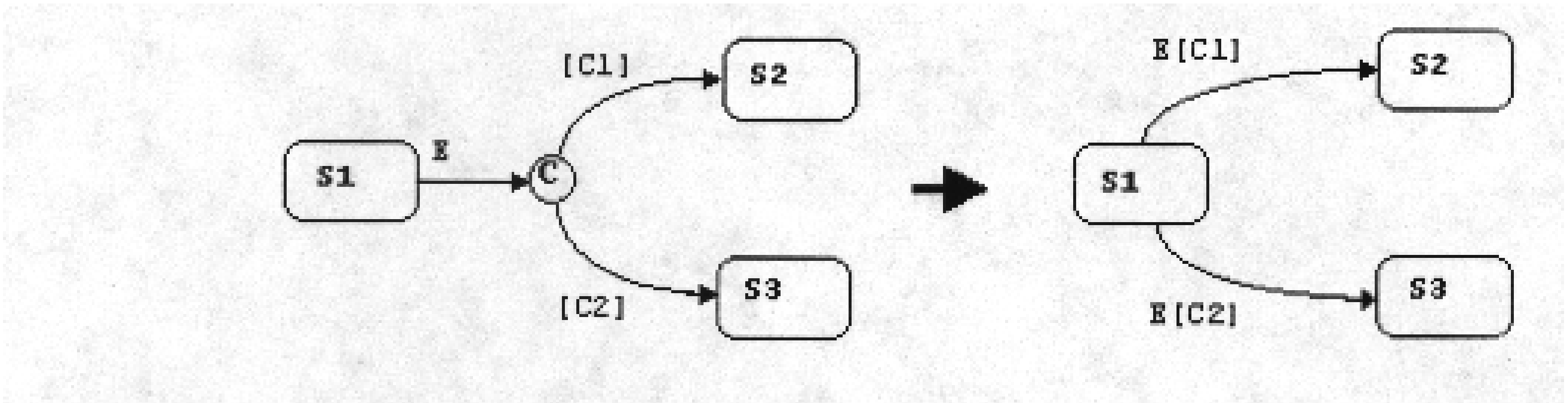


Figure 4.17. A condition connector and compound transitions

Switch Connector

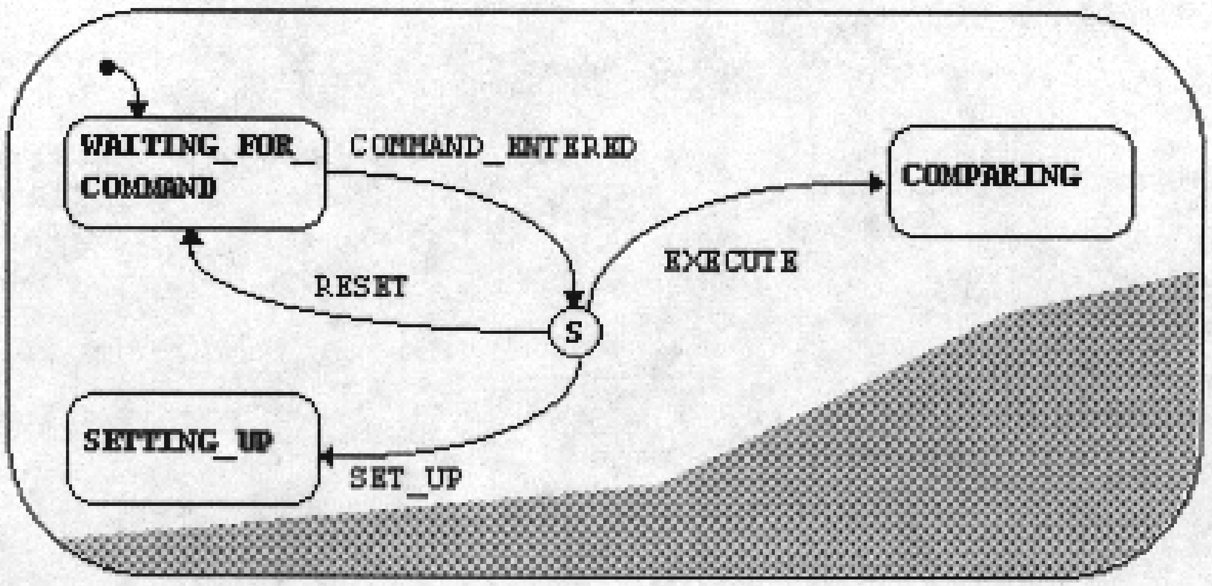
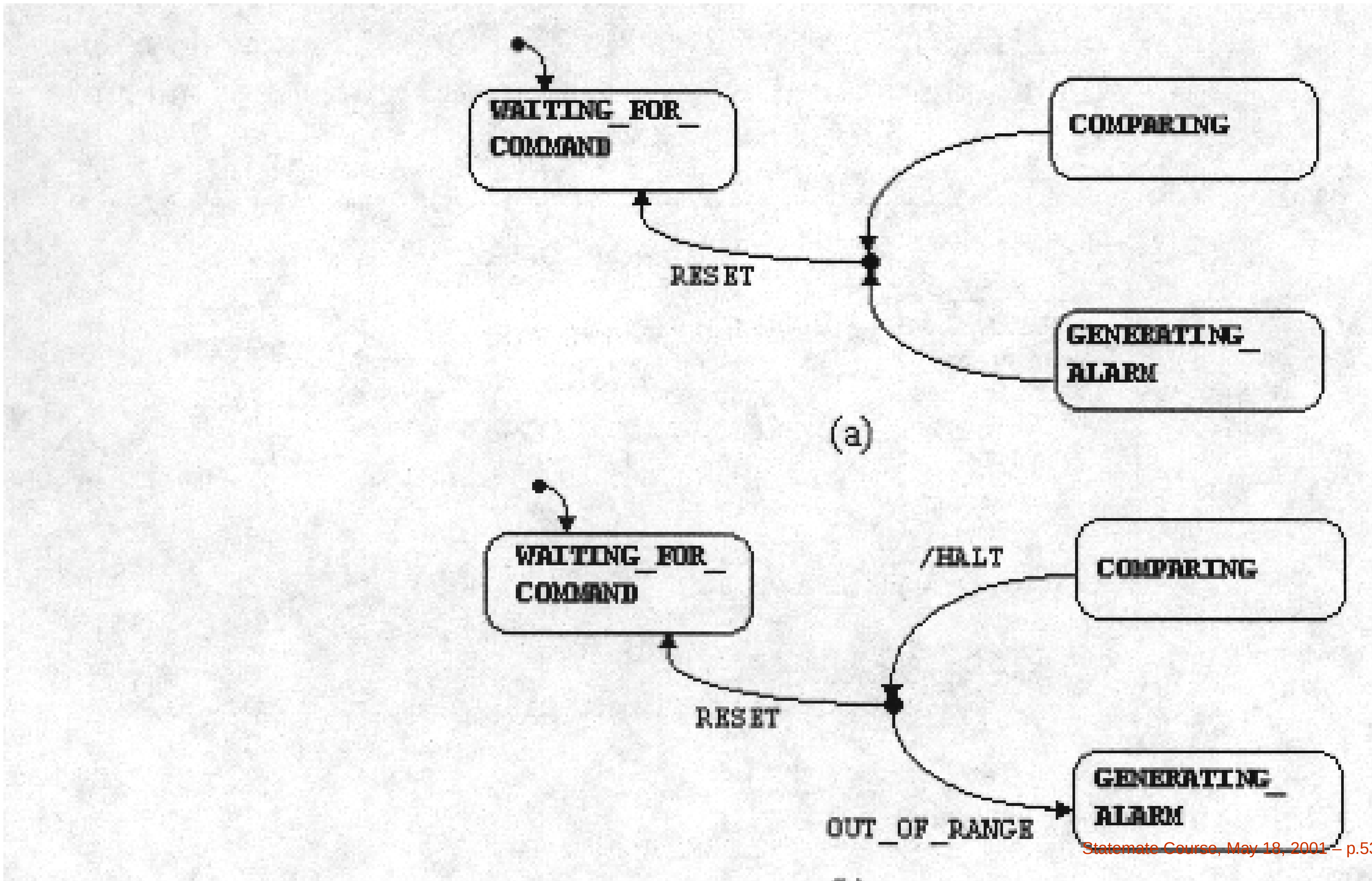


Figure 4.18. A switch connector

Junction Connector



Example

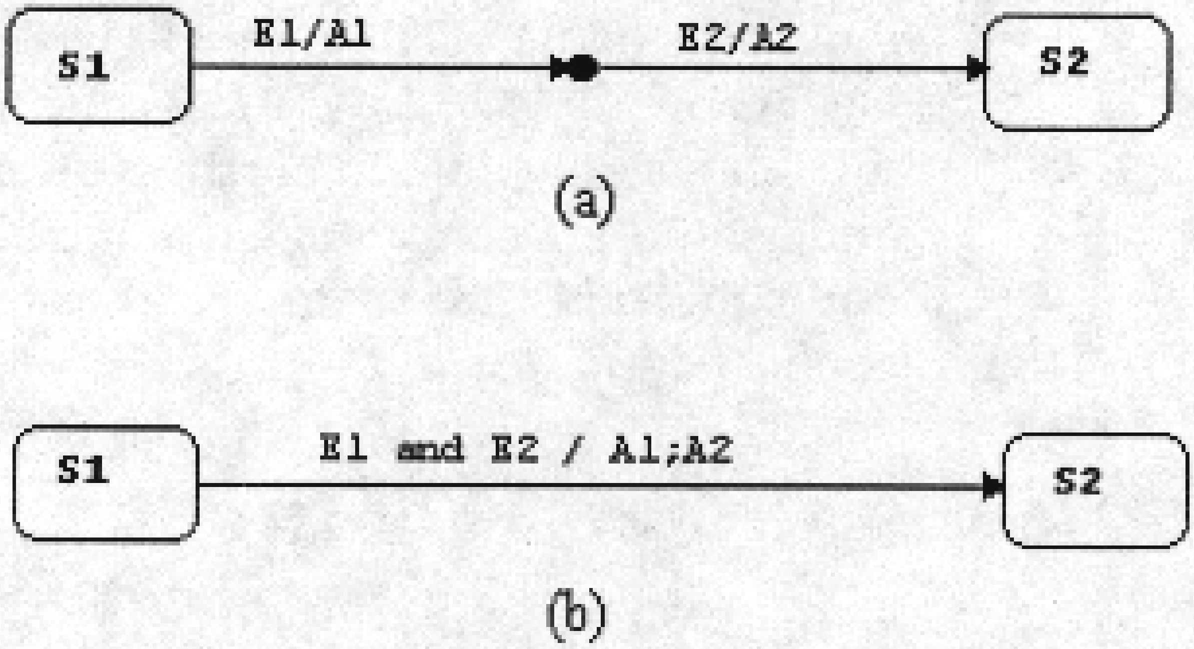


Figure 4.20. Two equivalent transition constructs

Diagram Connector

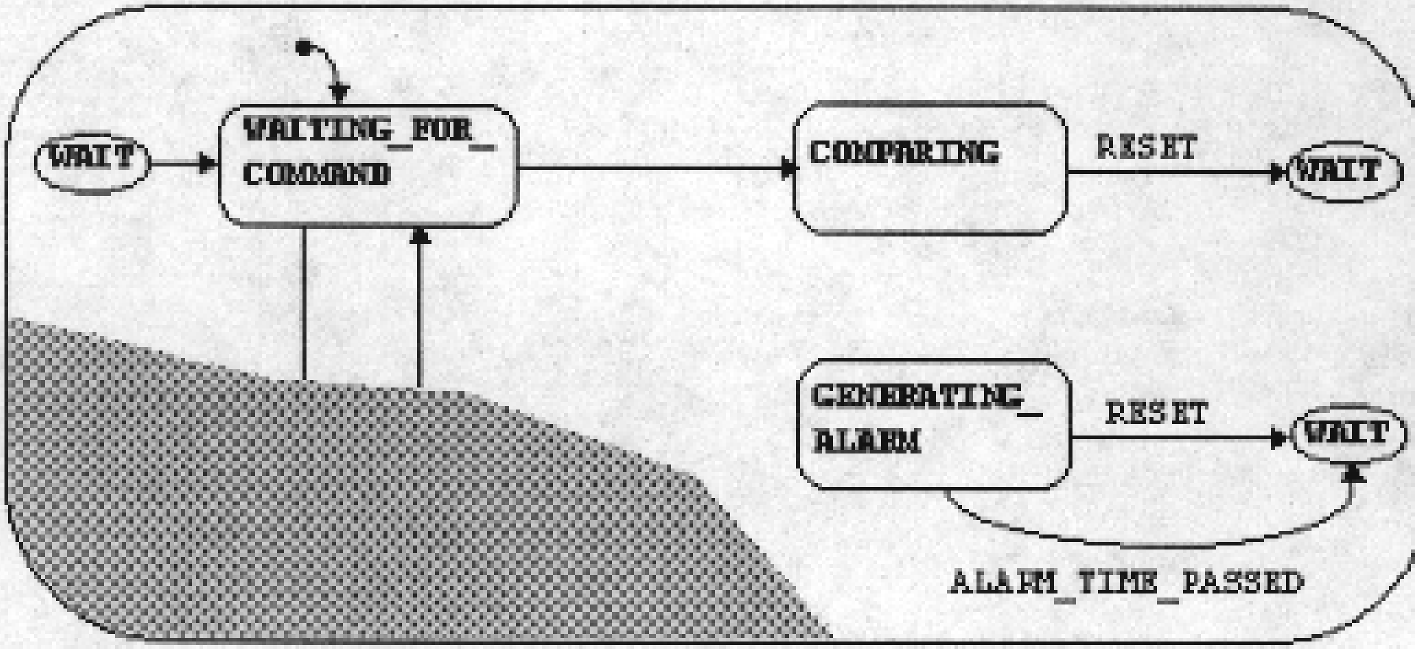


Figure 4.21. Diagram connectors

Transitions to and from And-States

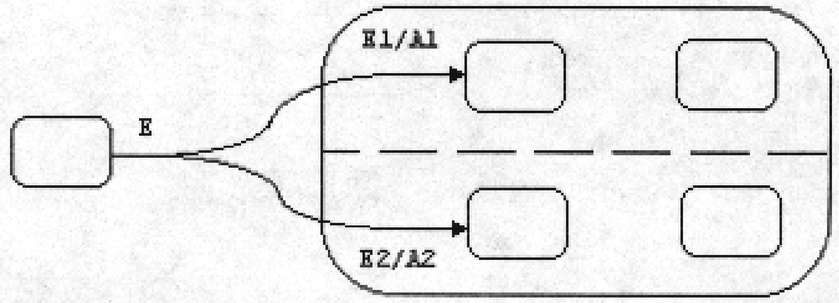


Figure 4.23. Triggers and actions on a fork construct

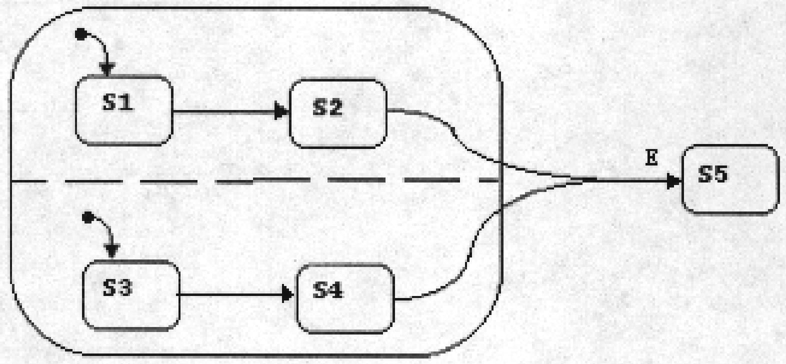


Figure 4.24. A joint connector in a merge construct

Asymmetric Cases

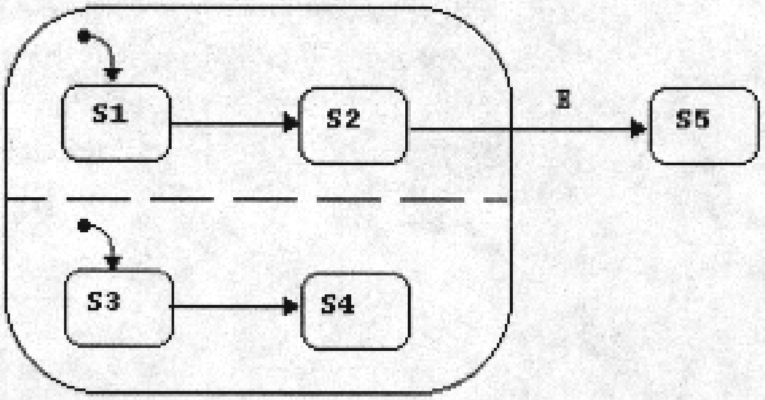
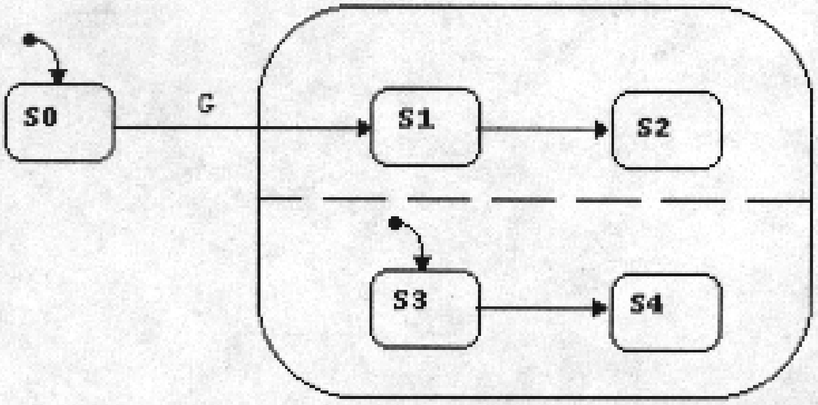


Figure 4.25. A transition from an and-state



Module-Charts

Structural Description: High-Level Design

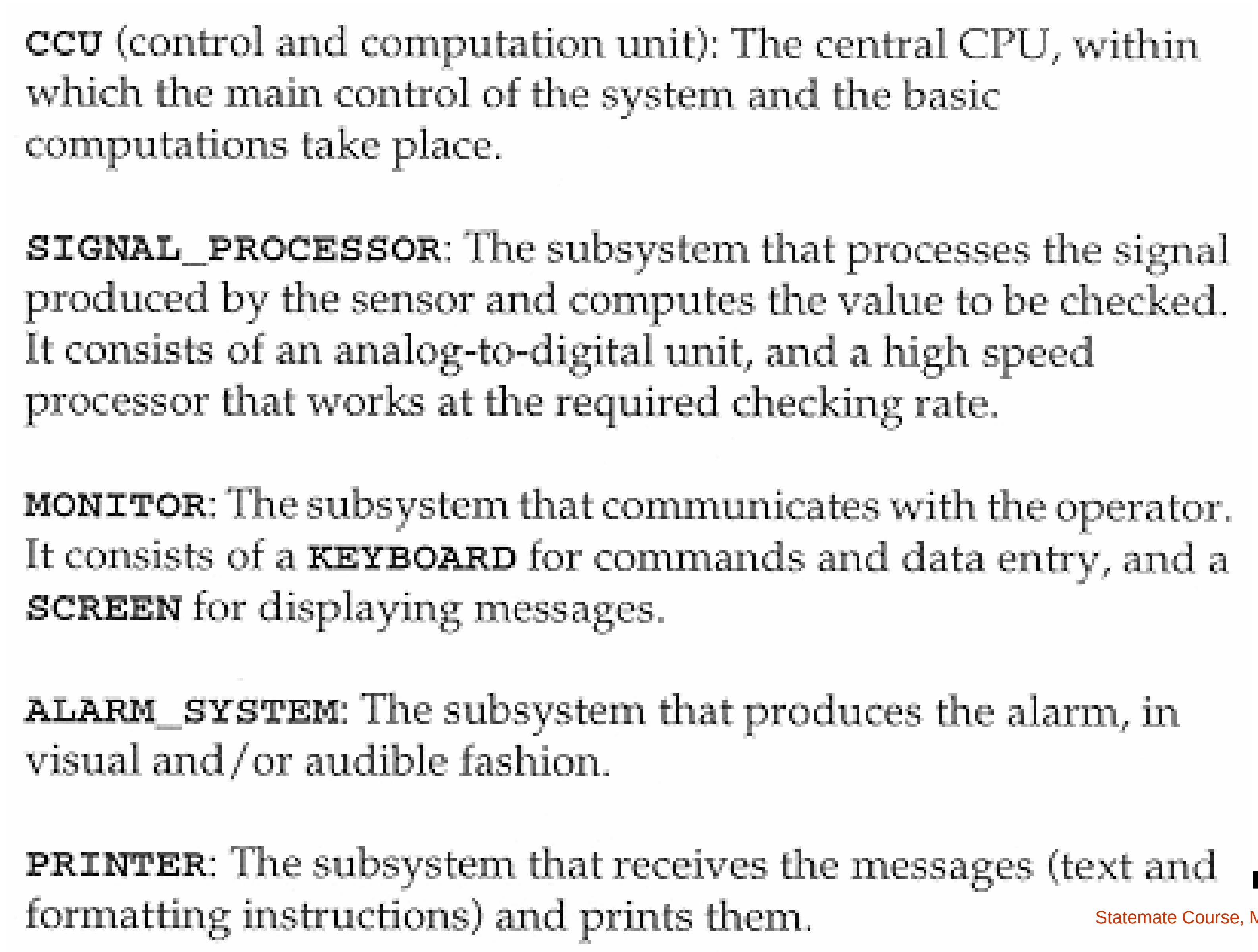
Module-charts describe the structural view – sometimes called the architectural view – of the system under development. Module-charts are typically used in the high-level design stage of the project.

Structural View

The structural view captures the system’s high-level design. A structural description of the system specifies the components that implement the capabilities described by the functional and behavioral views.
These components may be:

- hardware,
- software,
- or even humans.

EWS Example



CCU (control and computation unit): The central CPU, within which the main control of the system and the basic computations take place.

SIGNAL_PROCESSOR: The subsystem that processes the signal produced by the sensor and computes the value to be checked. It consists of an analog-to-digital unit, and a high speed processor that works at the required checking rate.

MONITOR: The subsystem that communicates with the operator. It consists of a **KEYBOARD** for commands and data entry, and a **SCREEN** for displaying messages.

ALARM_SYSTEM: The subsystem that produces the alarm, in visual and/or audible fashion.

PRINTER: The subsystem that receives the messages (text and formatting instructions) and prints them.

Connections to functional view

Sometimes There is a clear correspondence between the top-level activities in the functional view and the top-level subsystems in the structural view, e.g., SIGNAL_PROCESSOR implements the activity PROCESS_SIGNAL.
In other cases the structural decomposition is quite different from the functional decomposition. E.g., the CCU subsystem carries out both the EWS_CONTROL and COMPARE activities, whereas the DISPLAY_FAULT activity is divided into subactivities that are distributed among the ALARM_SYSTEM and MONITOR subsystems.

Internal and External Modules

The structural view is represented by the language of Module-charts.

- There exist two types of **internal** modules:
  - **execution modules**
  - **storage modules**
- And there exist **external modules**

Modules

- Execution modules may be submodules of other external modules only.
- Storage modules may be submodules of other storage modules or of execution modules.
- External modules are always external to an execution module or storage module, and there is no hierarchy of external modules.

EWS-Example

The next figure shows the structural decomposition of the EWS, including a storage module DISK, that stores the fault messages:

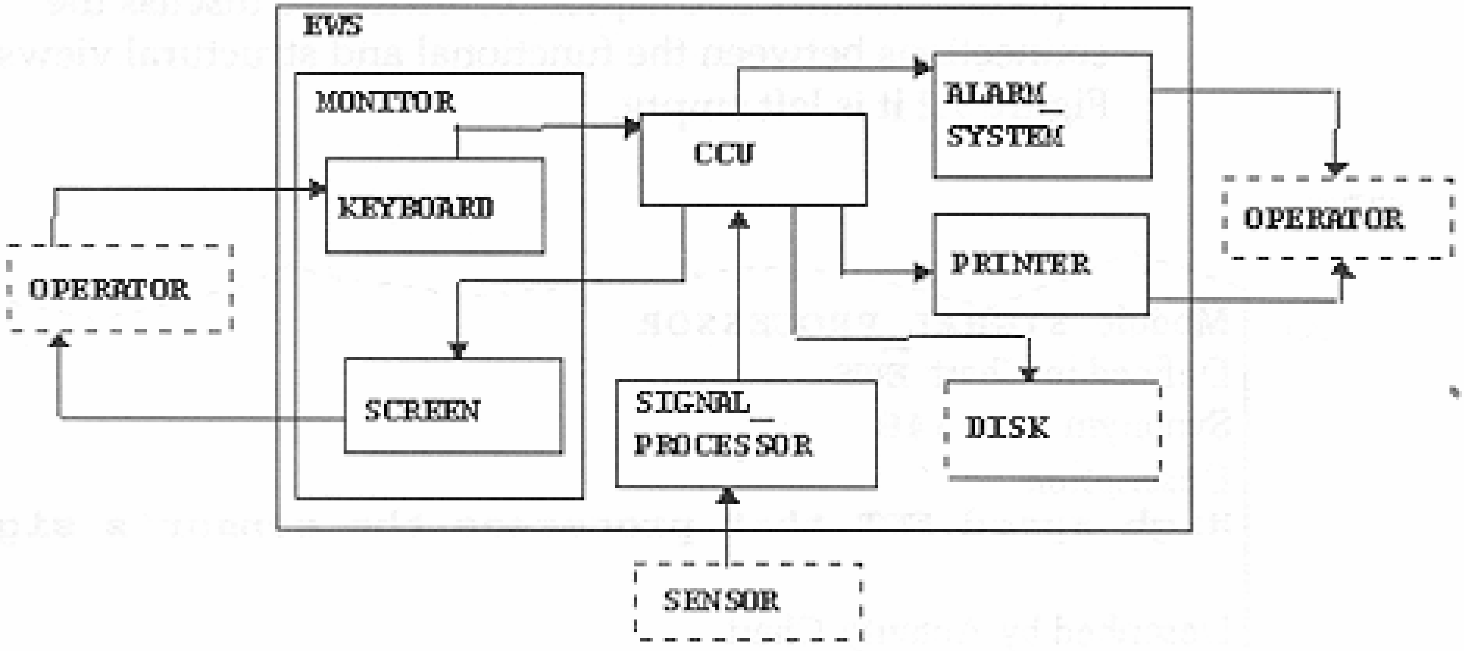
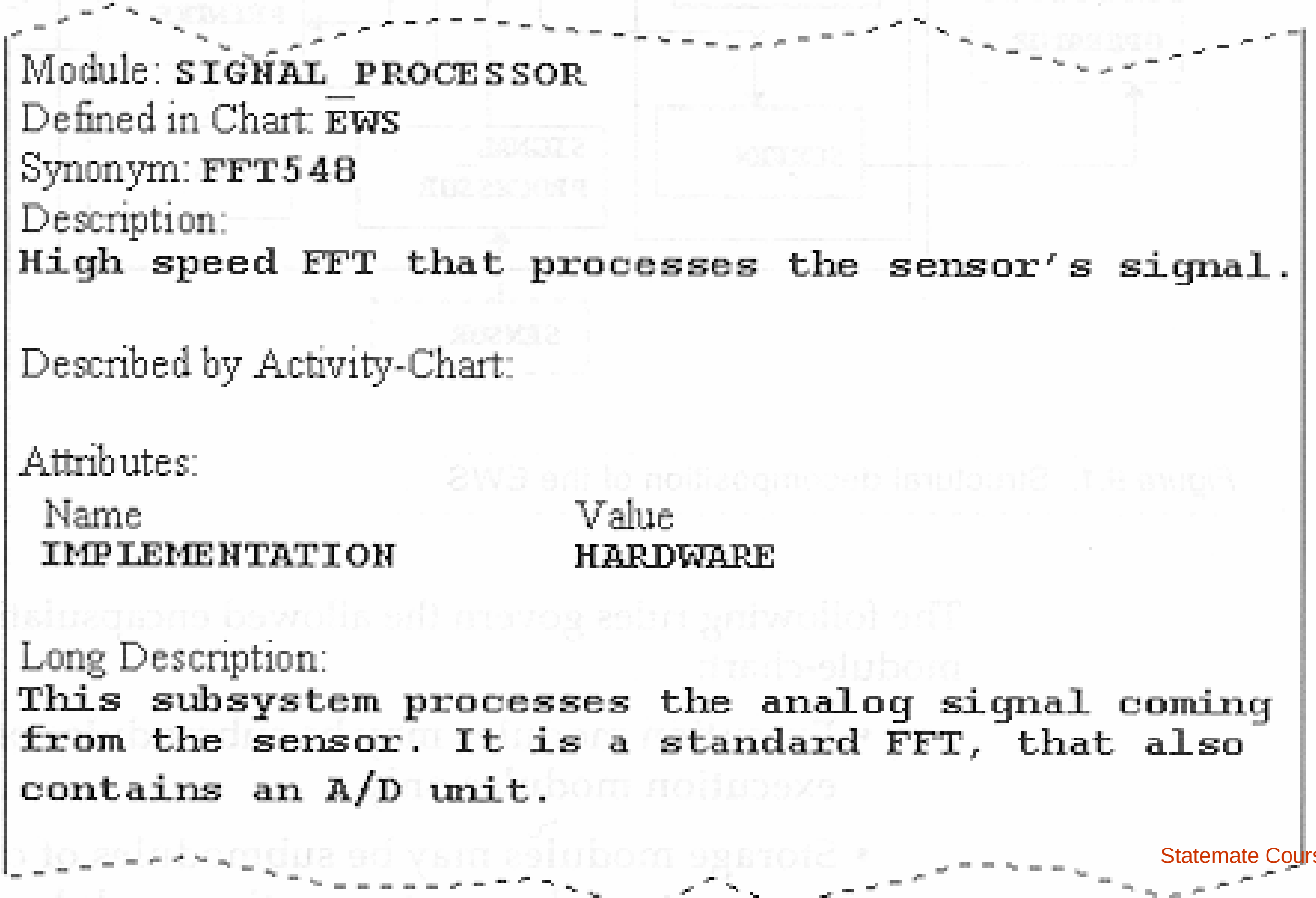


Figure 9.1. Structural decomposition of the EWS

Data Dictionary Entry

The Data Dictionary contains a special field, DESCRIBED BY ACTIVITY-CHART, which is used to connect modules with their functional description:



Module: SIGNAL_PROCESSOR
Defined in Chart: EWS
Synonym: FFT548
Description:
High speed FFT that processes the sensor's signal.

Described by Activity-Chart:

Attributes:

Name	Value
IMPLEMENTATION	HARDWARE

Long Description:
This subsystem processes the analog signal coming from the sensor. It is a standard FFT, that also contains an A/D unit.

Communication Between Modules

As in Activity-charts we use labeled arrows between modules to denote communication between them. They are called **flow-lines** or **m-flow-lines** to emphasize that they connect modules.

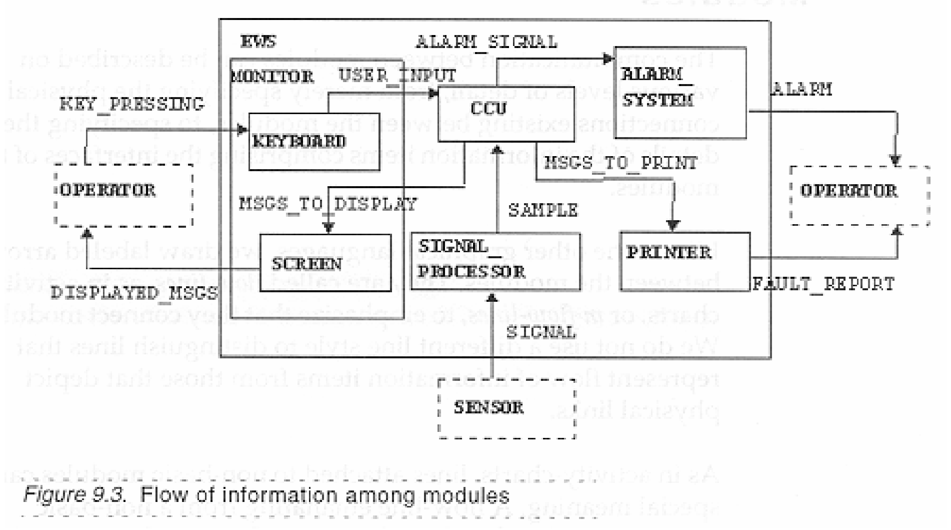


Figure 9.3. Flow of information among modules

Here, USER_INPUT contains the information-flow COMMANDS, the data-item RANGE_LIMITS and the condition SENSOR_CONNECTED.

Physical Links Between Modules

Arrows in a module-chart may also denote physical communication links, or channels, between modules:

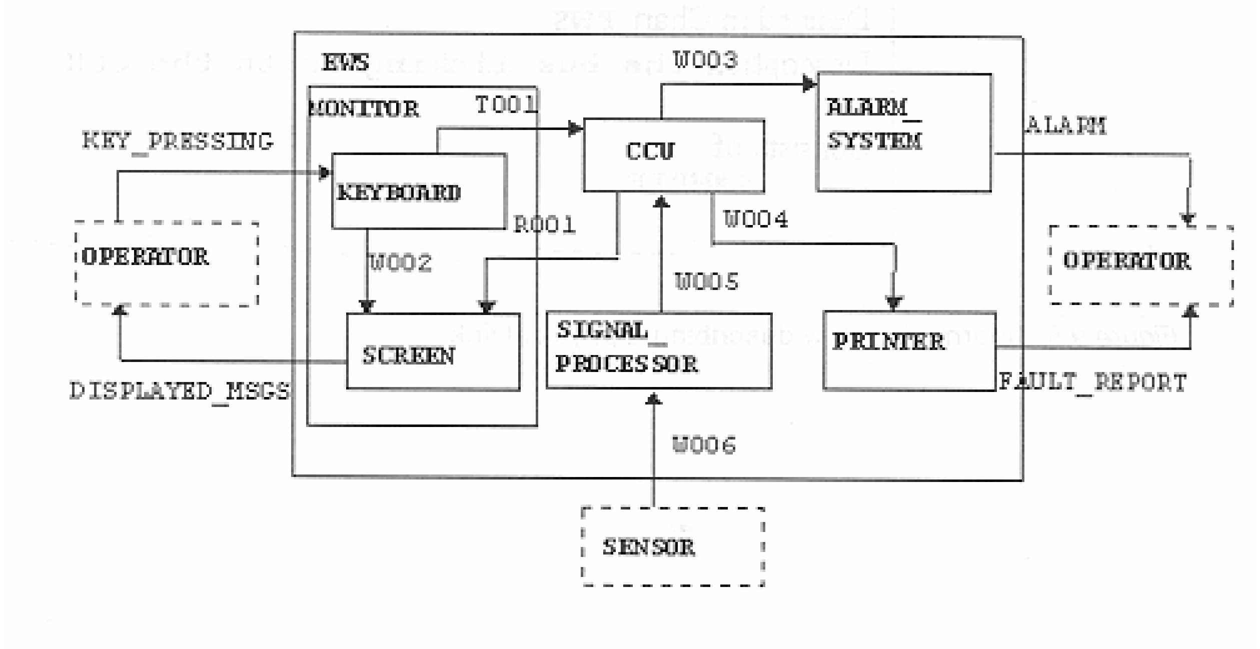
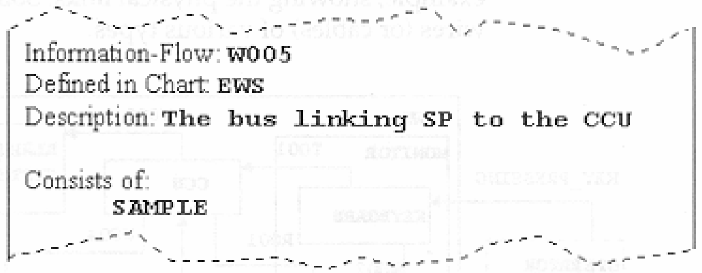


Figure 9.4. Physical links among modules


Information-Flow: W005
Defined in Chart: EWS
Description: The bus linking SP to the CCU

Consists of:
SAMPLE

Connectors and Compound Flow-Lines

Connectors and compound flow-lines are allowed in module-charts exactly as in activity-charts:

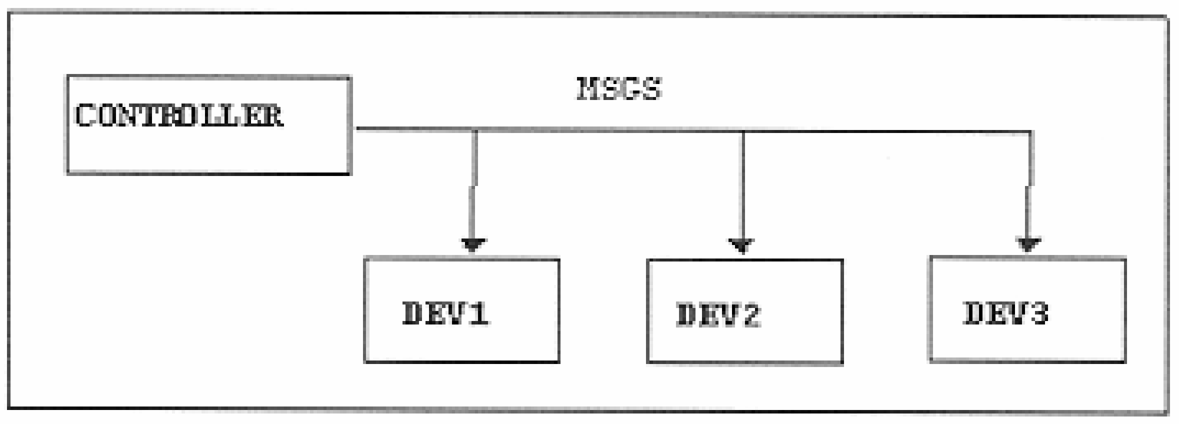


Figure 9.6. Communication link to several devices

Connections Between the Functional and Structu

- The functional view provides a decomposition of the system under development into its functional components, i.e., its capabilities and processes.
- The structural view provides a decomposition of the system into the actual subsystems that will be part of the final system, and which implement its functionality.

Types of connections

There are three types of connections between the functional and structural views:

1. describe the functionality of a module by an activity-chart: **Activity-chart Describing a Module**
2. allocate specific activities in an activity-chart to be implemented in a module: **Activities Implemented by Modules**
3. map activities in the functional description of one module to activities in that of some other module: **Activities Associated with a Module's Activ ities**

Conclusion

In conclusion, we may wish to attach functional descriptions, i.e., activity-charts, to modules on different levels of the structural decomposition:

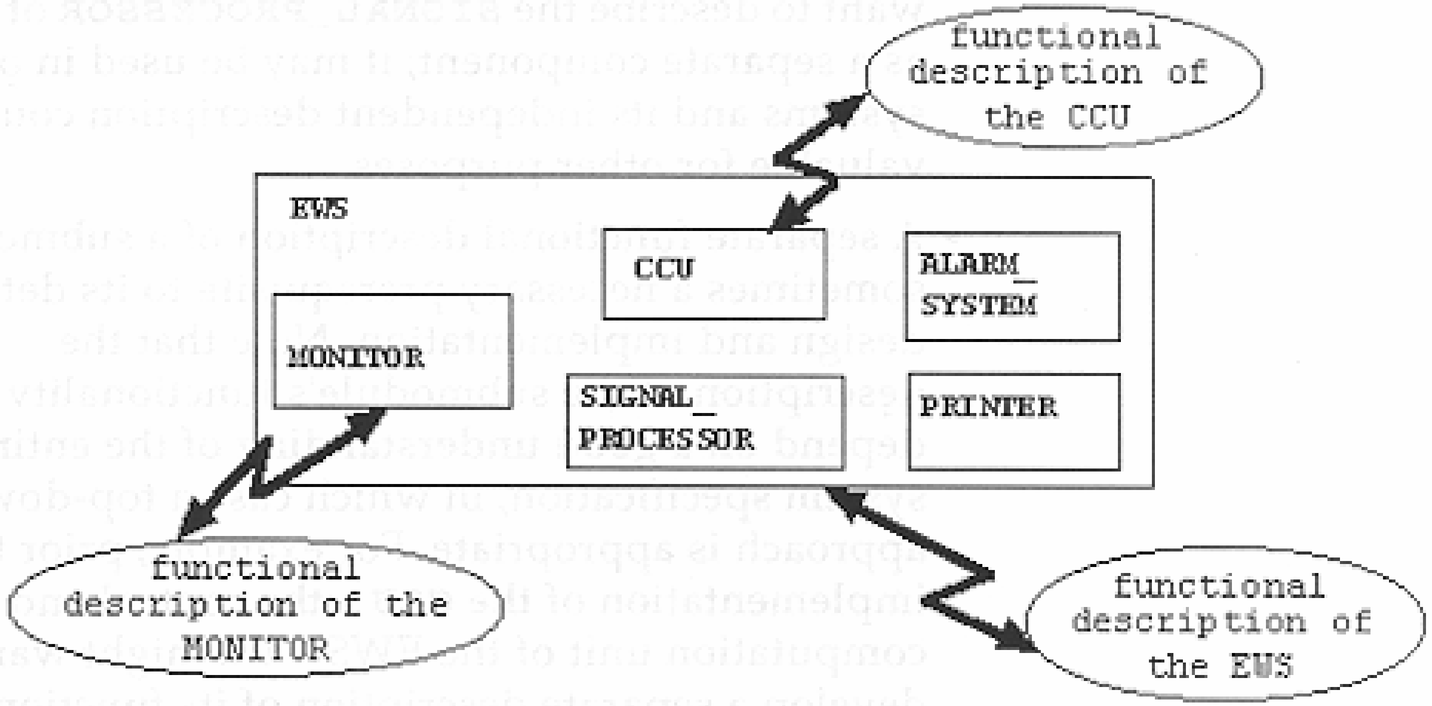

functional description of the CCU
EWS
CCU
ALARM_ SYSTEM
MONITOR
SIGNAL_ PROCESSOR
PRINTER
functional description of the MONITOR
functional description of the EWS

Figure 10.1. Functional descriptions attached to different modules

Activity-chart Describing a Module

The activity-chart EWS_ACTIVITIES

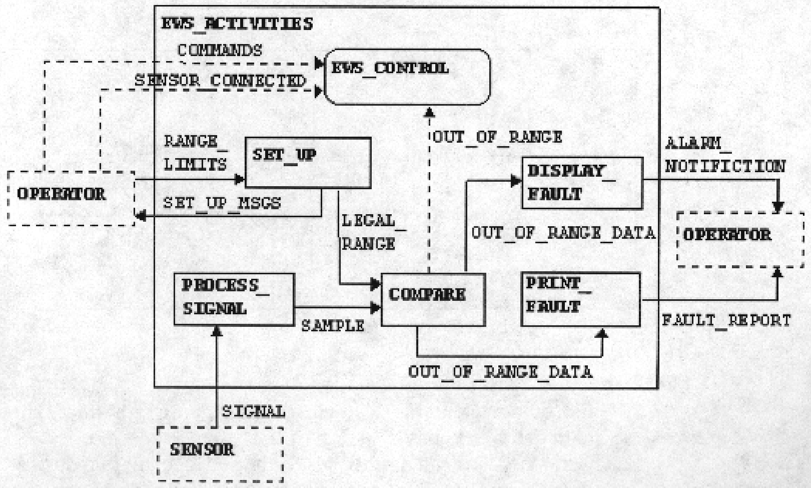
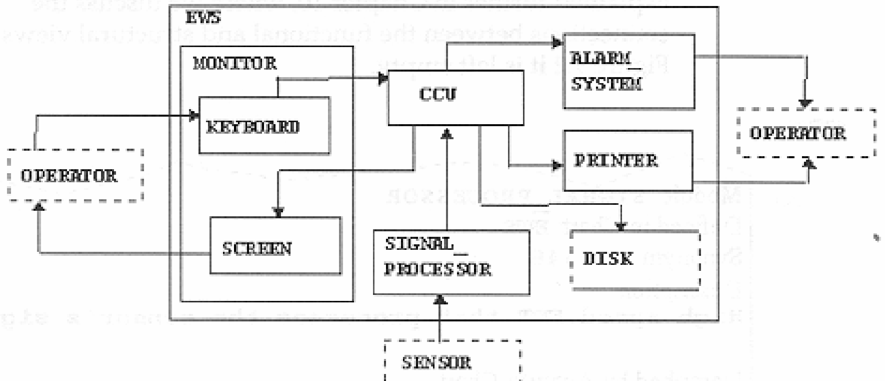
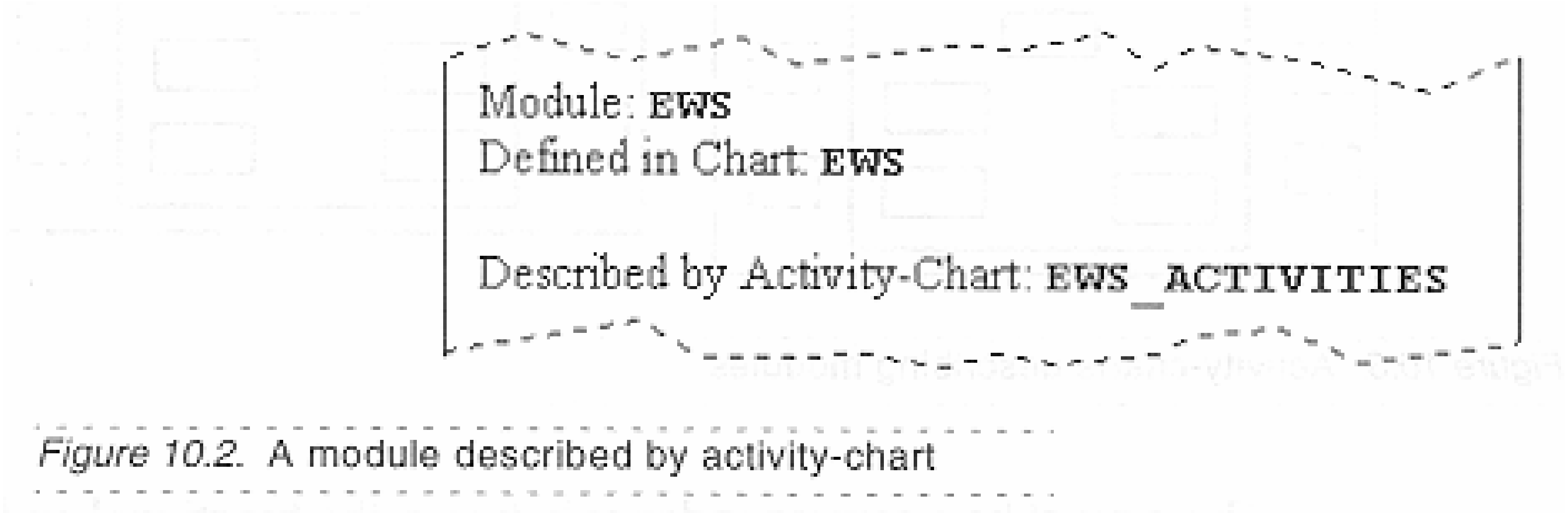


Figure 2.5. **EWS_ACTIVITIES**, its environment and flow of information

describes the functionality of the module EWS



This connection is specified in the Data Dictionary:



Module: EWS
Defined in Chart: EWS

Described by Activity-Chart: EWS_ACTIVITIES

Figure 10.2. A module described by activity-chart

Notice that the connection is between an activity-chart and a module!

Top-Down Approach

One may now want to specify an activity-chart CCU_AC for the module CCU:

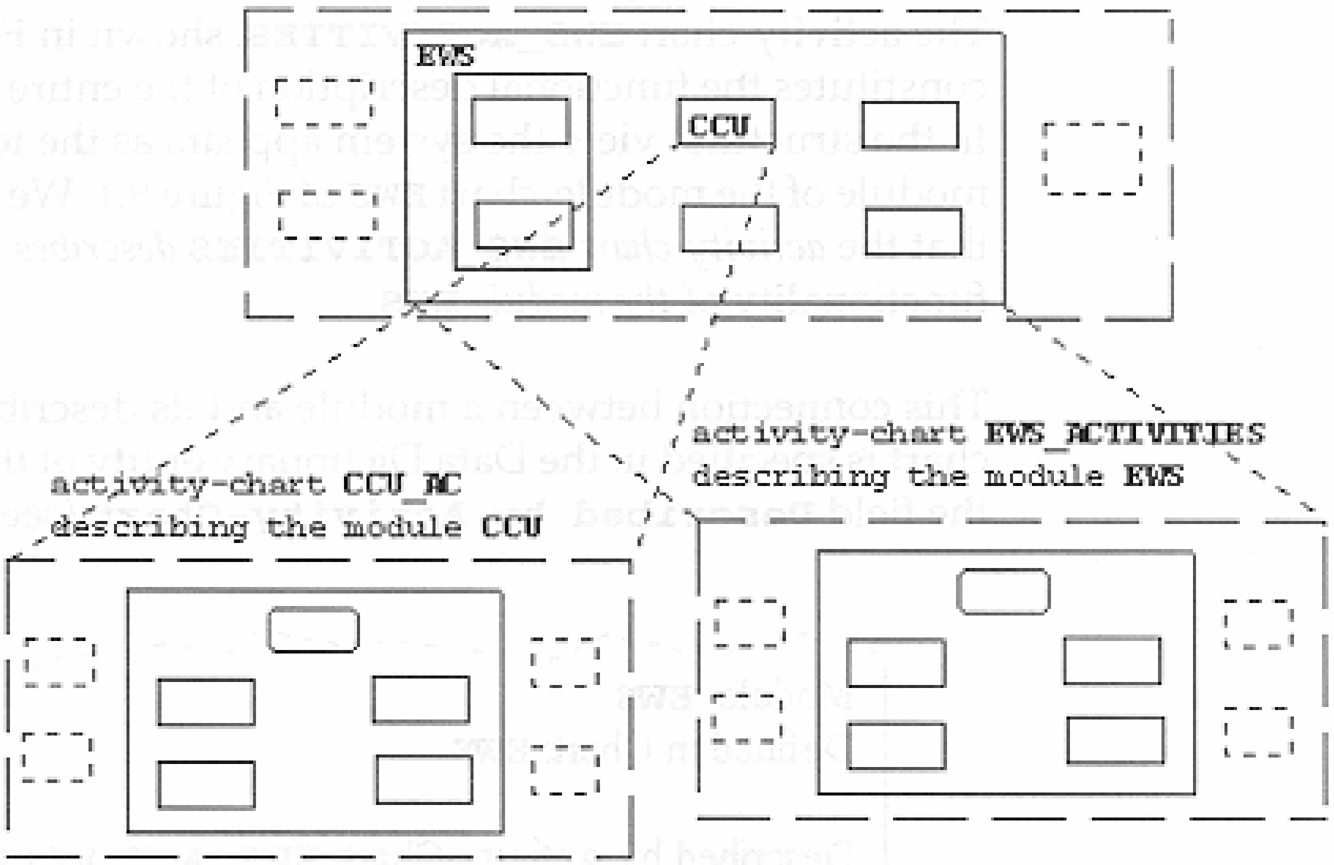
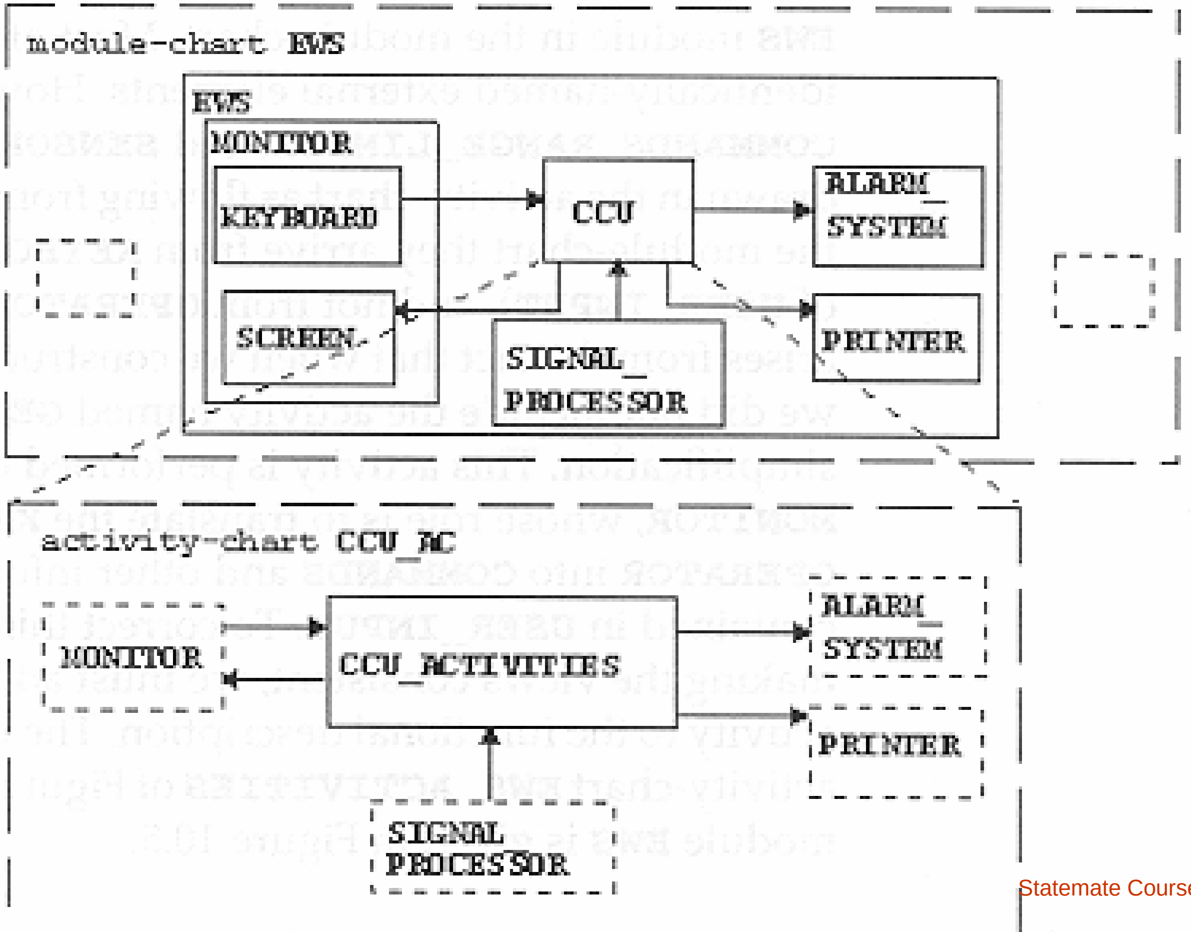

EWS
CCU
activity-chart EWS_ACTIVITIES
describing the module EWS
activity-chart CCU_AC
describing the module CCU

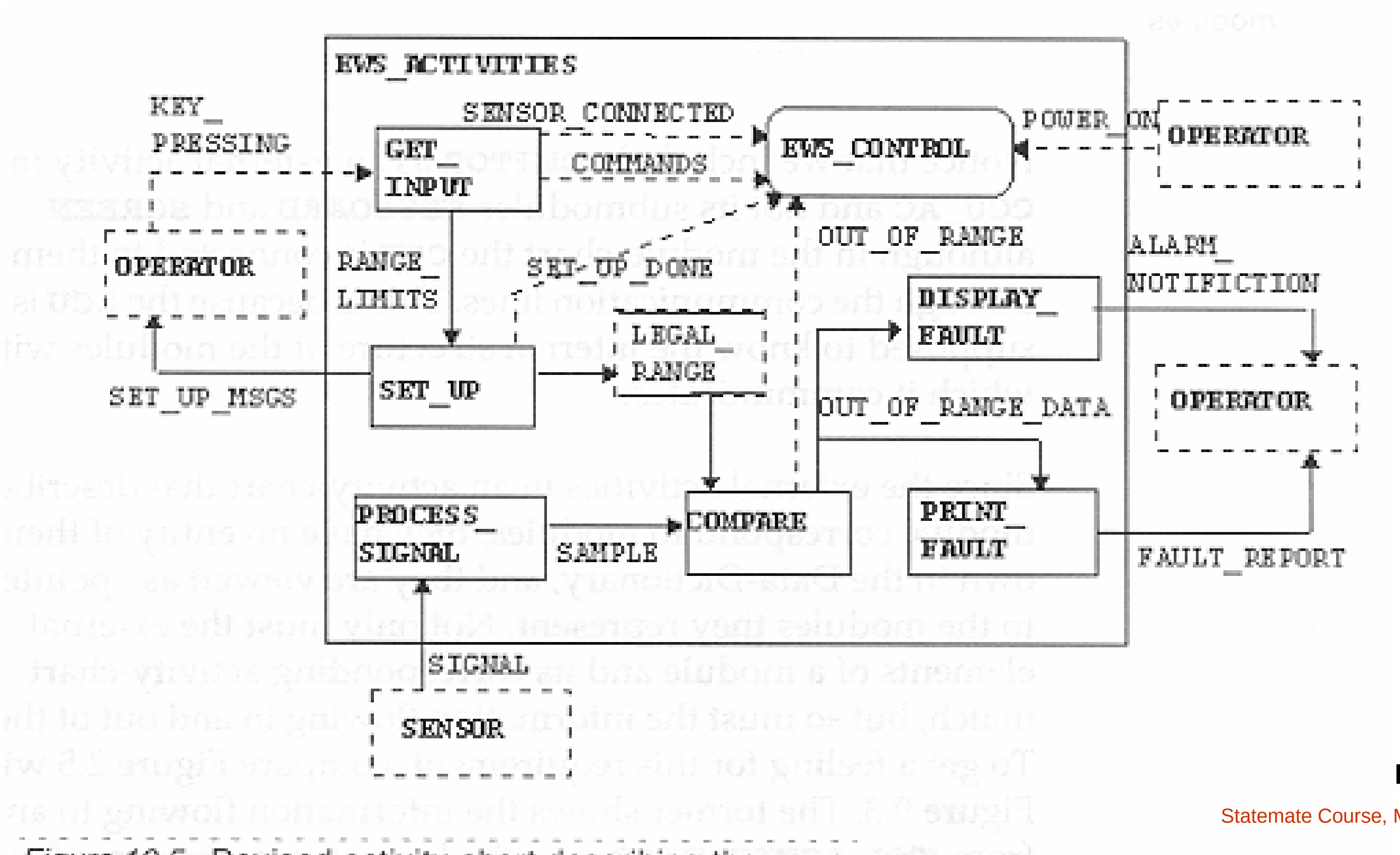
Figure 10.3. Activity-charts describing modules

Correspondence between views

There must be a correspondence between the functional and structural decompositions of a module in terms of the environment and the interface with it:

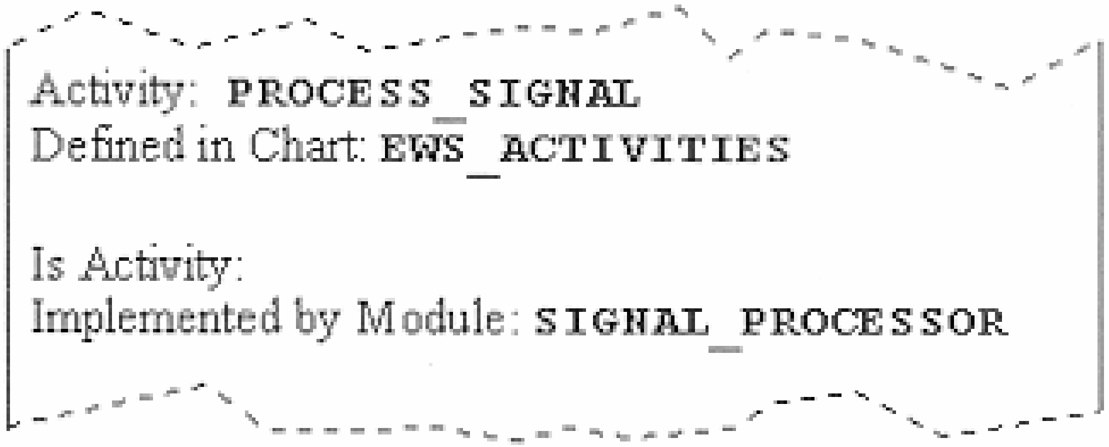


Since also the flow-lines have to be correct we have to introduce an activity GET_INPUT which will be implemented by the MONITOR module:



Activities implemented by Modules

When the module described by the activity-chart is eventually decomposed into submodules, we may be more concrete and allocate the relevant activities and data-stores to the submodules:

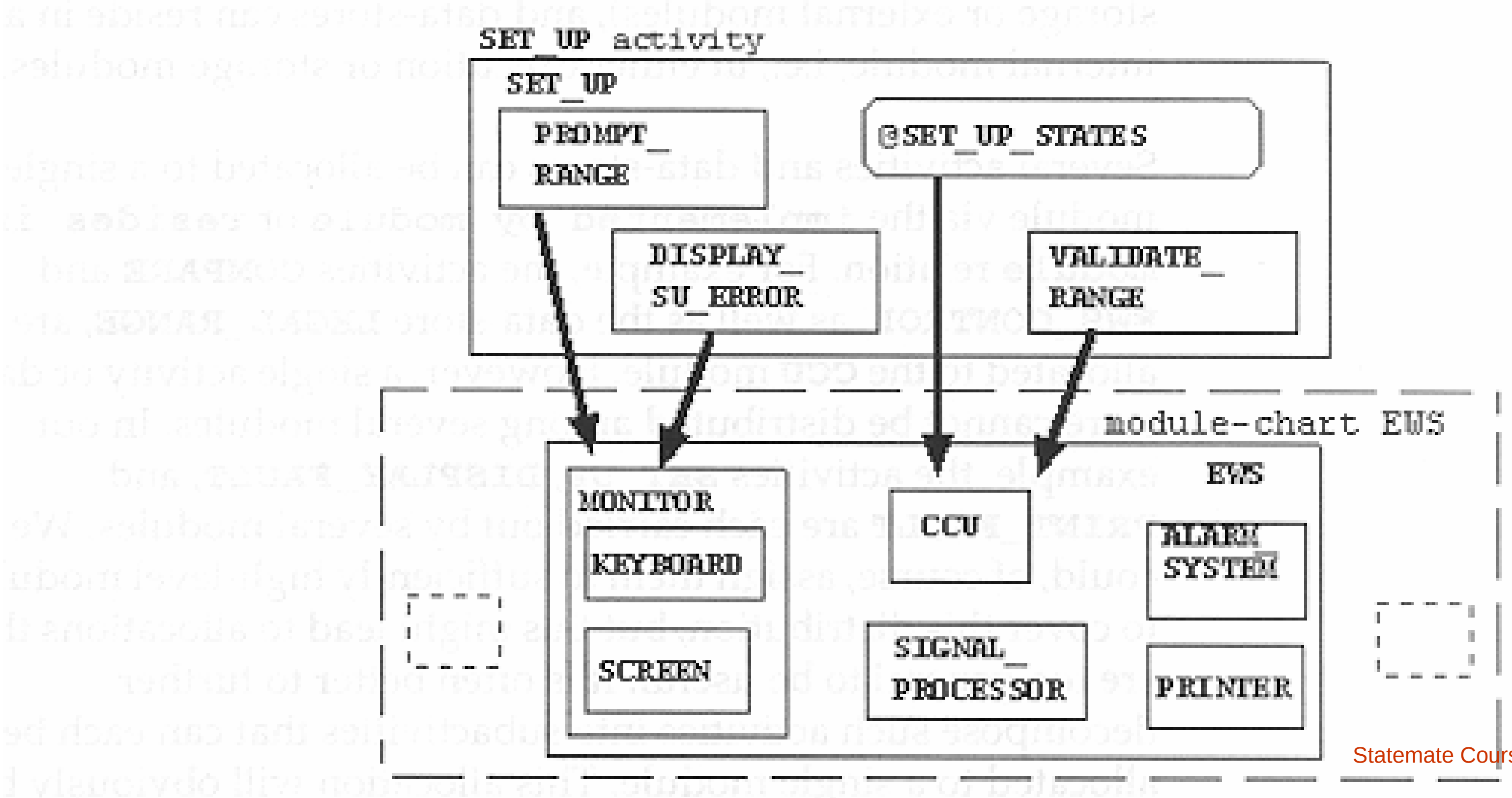


Activity: PROCESS_SIGNAL
Defined in Chart: EWS_ACTIVITIES

Is Activity:
Implemented by Module: SIGNAL_PROCESSOR

Figure 10.6. An activity implemented by a module

A single activity or data-store cannot be distributed among several modules.
Therefore, one has to decompose such activities (or data-stores) into subactivities that can each be allocated to a single module:



Activities Associated with a Module's Activities

On the one hand, there is the EWS_ACTIVITIES describing the functionality of the whole system. On the other hand, also the submodules implement activities:

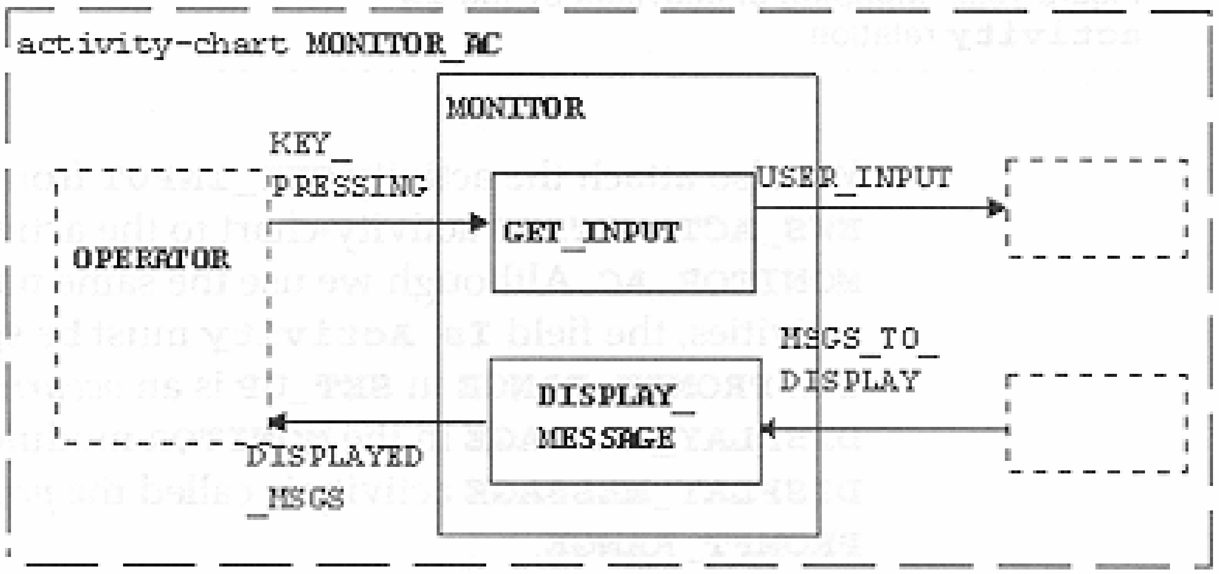
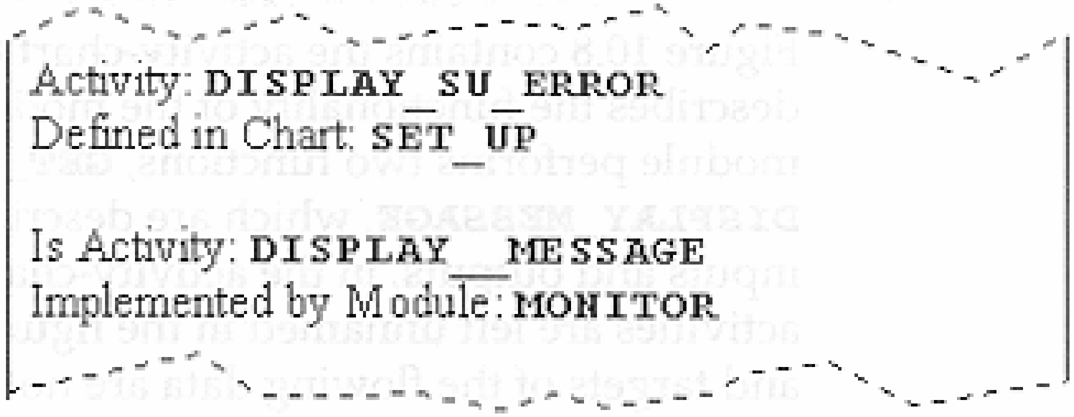


Figure 10.8. Activity-chart of MONITOR

Then, one wishes to associate subactivities of EWS_ACTIVITIES with those implemented by a submodule:


Activity: **DISPLAY_SU_ERROR**
Defined in Chart: **SET_UP**

Is Activity: **DISPLAY_MESSAGE**
Implemented by Module: **MONITOR**

Figure 10.9. Mapping of activities by the **is activity** relation