Statemate Course

Statemate/SDL — Teleteaching Vorlesung

W.-P. de Roever
K. Baukus
CAU Kiel

D. Hogrefe
H. Neukirchen
MU Lübeck

Session I

Introduction to Statecharts

Everything you wanted to know about Statecharts
but were afraid to ask

Abstract: The notion of a reactive system and the language Statecharts are introduced. The rationale behind the design decisions of Statecharts is explained in relation to the specific nature of reactive systems.

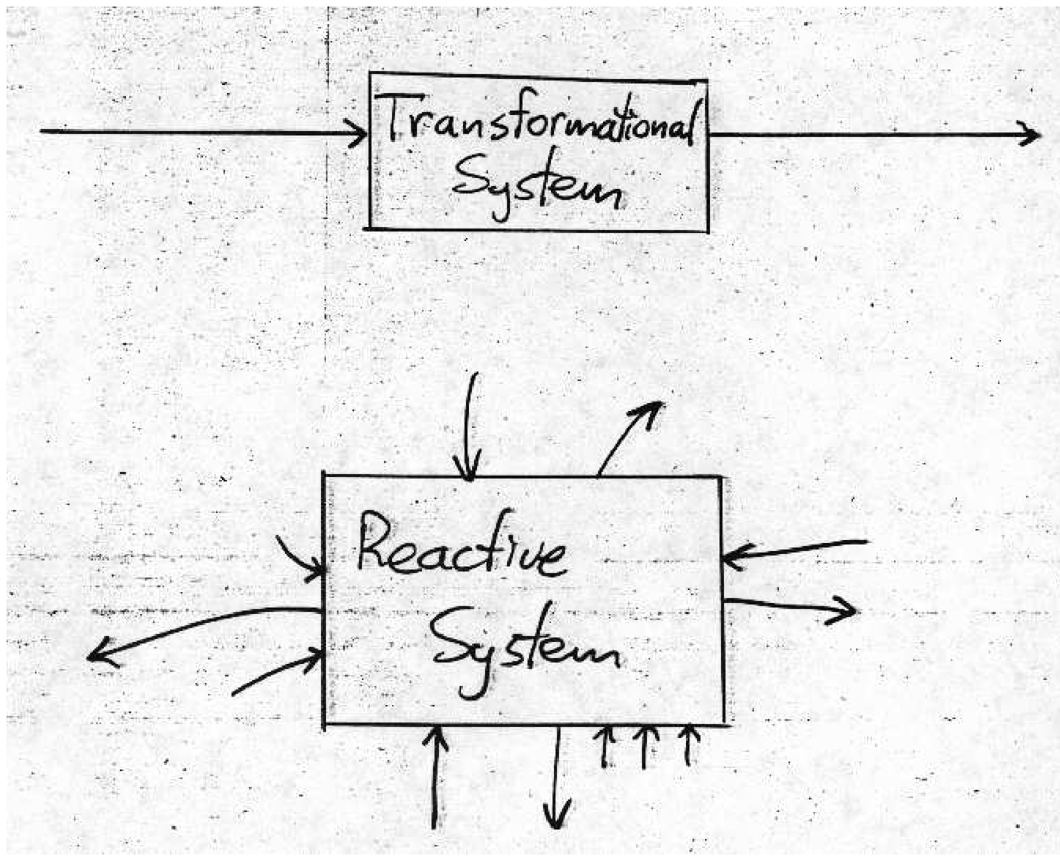
Literature: Introduction to Design Choices in the Semantics of Statecharts, C. Huizing, W.-P. de Roever, Information Processing Letters 37, p. 205-213, 1991

0.1 What are reactive systems?

- There is a fundamental dichotomy in the analysis of reactive systems; namely, the dichotomy between
  - **transformational** and
  - **reactive systems**.
- **Transformational systems** are described by the relation between initial and corresponding final states; they have a linear structure, because only the initial and corresponding final states are of interest.

  Examples include: sorting algorithms, compilers, and other algorithms computing a function as discussed in your data structures and complexity of algorithms course.
- **Reactive systems** do not compute a function, but are in continuous **interaction** with their environment.

  Examples: your tv set, digital watches, chips, interactive software systems, game programs s.a. trackman, monkey island, tomb raider, and other interactive computer games, but also one's heart monitor at an intensive care unit.
- Transformational systems are well-studied; for their programming and analysis many good languages and theories exist.
- We explain why the language Statecharts is a good candidate for **specifying** and **programming** reactive systems.


Transformational System
Reactive System

0.2 Why not use transformational description techniques?

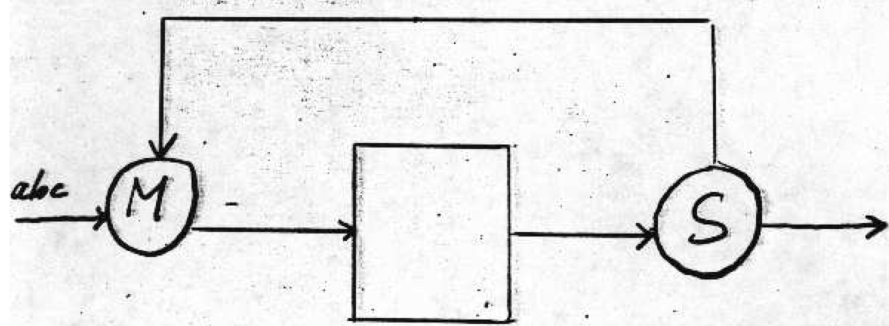
- If transformational systems are so well studied, why doesn't one consider a reactive system as a transformational one?

  Simply say that a reactive system transforms a **sequence** of inputs to a sequence of outputs.
- This doesn't work, because of "feedback", as illustrated by the **Brock-Ackermann** paradox.
- Consider two systems, a one-place buffer and a two-place buffer. If you consider these transformationally, they display the same initial-final state behavior.

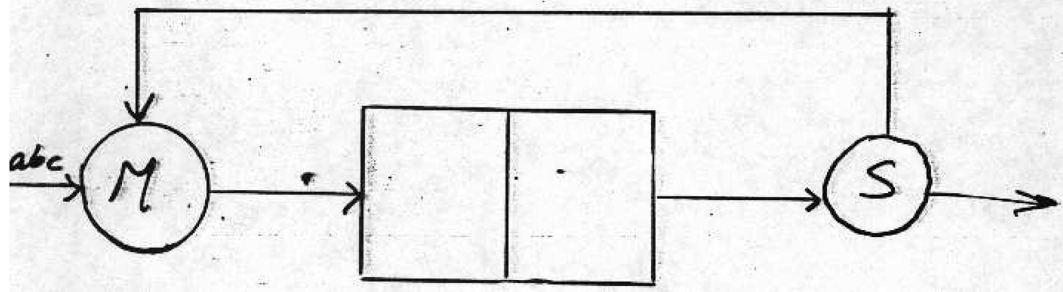
  But if the output of these systems is fed back, and merged with their input they behave differently. (See transparency)
- **Conclusion:** The relative order of output events relative to the input events needs to be specified, in order to characterize the semantics of a system with interaction with its environment through feedback.
  (One needs to know when an output is produced)

Brock-Ackerman paradox

1. One-place Buffer



2. Two-place Buffer



in: $abc\ldots$
out: 1. e.g. $abac\ldots$
2. e.g. $abca\ldots$
not $abac\ldots$

0.3 Graphical languages

- Transformational systems have a linear structure, and so have the conventional languages for specifying and programming them. What one describes is how a final state is produced from an initial one. **The relative time when intermediate states are computed is not important**, and neither is their identity as long as the corresponding final state is known!

- For reactive systems this is completely different:
  - The “moment” a new input arrives is relevant to the behavior of the system $\Longrightarrow$
  - The internal state of the system at the time of input is important for the systems reaction.
  - Reactive systems may not even have a final state!

- So, in reactive systems **there is no main sequential flow of control** (as in transformational systems) and **statements can have several entry and exit points**.

Graphical formalisms for reactive systems

There exist graphical formalisms for describing reactive systems:

- E.g., state diagrams for **finite state machines (FSMs)**:

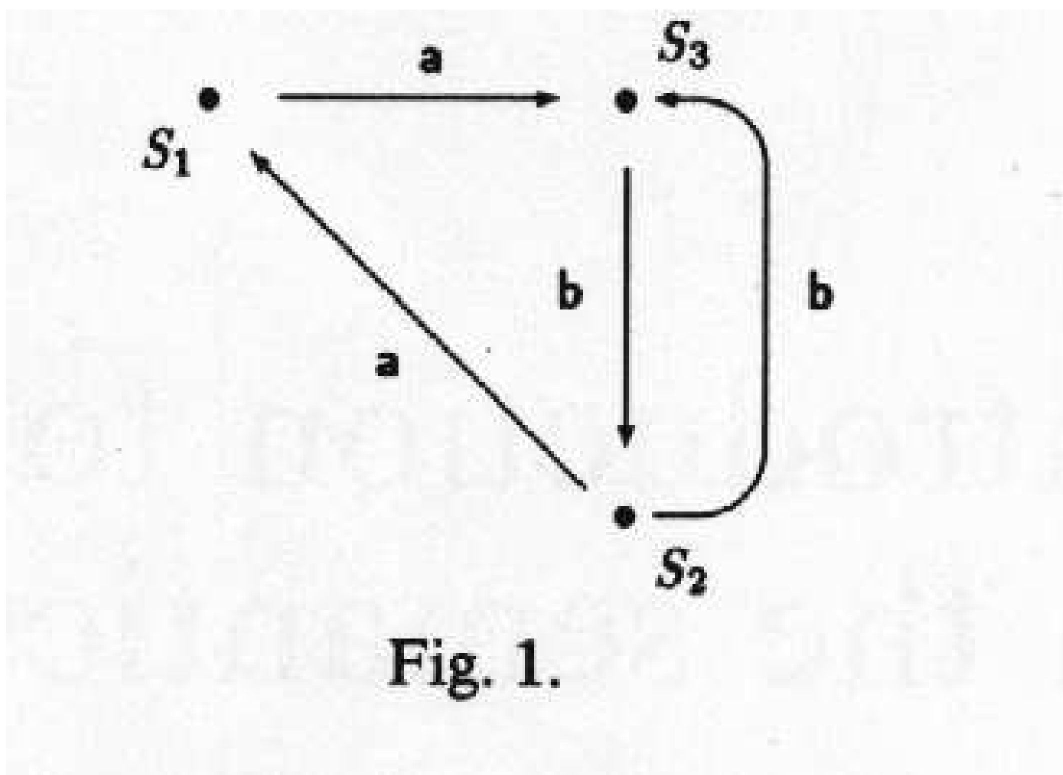


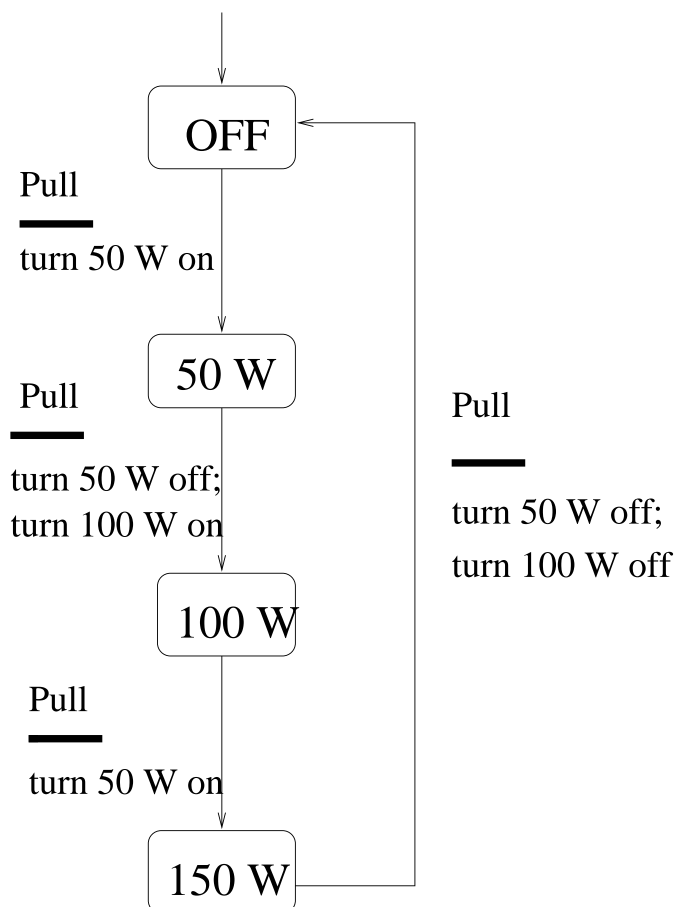
Fig. 1.

For each state, the possible reactions to input that arrives in that state is specified by a transition to other states.

- However, these don't serve our purpose: The only output a FSM produces is a signal that it has reached its final state.

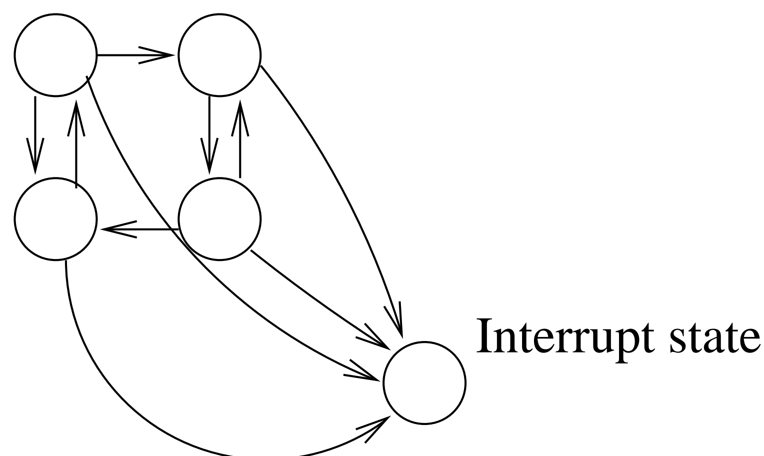
  A reactive system, however, may produce output at any time of its execution $\Longrightarrow$
- Use **Mealy machines**: these can produce an output whenever a transition is made.

  Example: A lamp having two bulbs — one of 50 W and one of 100 W — which can be lighted separately and together. Note that in this example the data flow $\neq$ control flow:



Disadvantages of Mealy machines for describing reactive systems

- **They have no structure**. There is no strategy for their top-down or bottom-up development.
- They are not economical w.r.t. transitions, when one event has all transitions as a starting point as in case of interrupts:



- They are not economical w.r.t. **parallel composition**: **Exponential growth** in the number of states when composed in parallel.

$\Longrightarrow$ We need a formalism for the hierarchical development and refinement of Mealy machines.

This is provided by Statecharts, invented by David Harel.

Statecharts display **hierarchy** and **structure**, and enable hierarchical development.

0.4 Hierarchy and Structure

The concepts of hierarchy and structure in Statecharts are introduced using a quite familiar example of a reactive system: that of a **television set with remote control**.

0.4.1 First concept: Hierarchy

Hierarchy or depth in states, and interrupts. This is achieved by drawing states as boxes that contain other boxes as sub-states.

- The television set can be in two states: **on** and **standby**. Switching between them is done by pushing the **on** and **off** buttons, generating the **on** and **off** events:

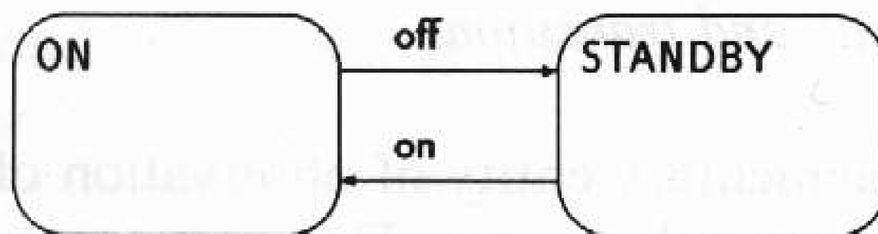


Fig. 2.

- In state **on** the tv set can be in two sub-states: **normal** and **videotext**:

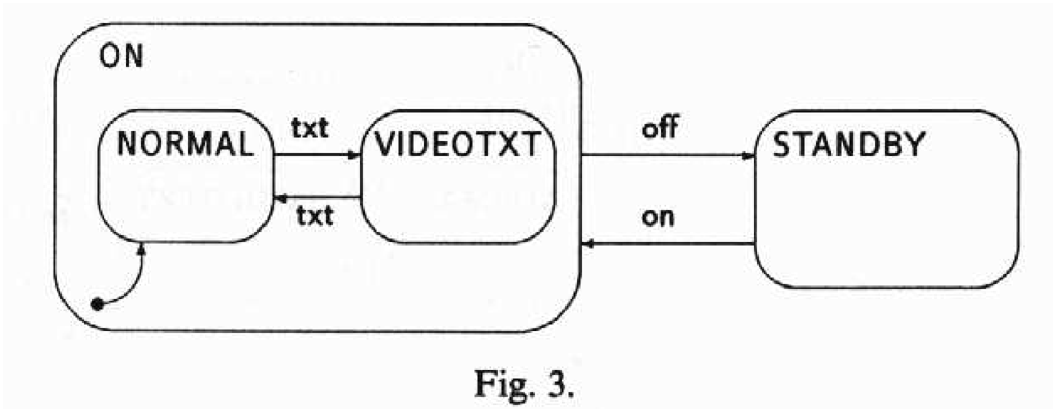


Fig. 3.

The ⟶ arrow leading to **normal** specifies which sub-state should be entered when the higher level state**on** is entered, namely **normal**.

- When in **on** an event **off** is generated, this acts like an interrupt and state **on** (incl. all its sub-states) is left, and control switches to state **standby**. In this way interrupts are handled without cluttering the picture with arrows as on transparency no. 9.

0.4.2 Second concept: Orthogonality

- Two independent components can be put together into an **AND-state**, separated by a dotted line

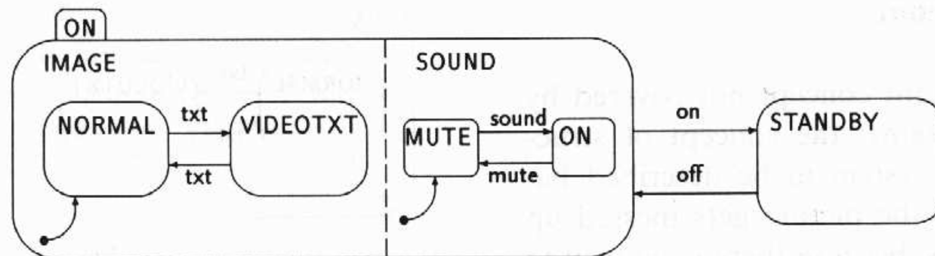


Fig. 4.

- Being in an AND-state means being in all of its immediate sub-states **at the same time**. This prevents the exponential blow-up familiar from composing FSMs in parallel.

0.4.3 Third concept: Broadcast

- In our case we split state **normal** in two orthogonal components **channel**, for selecting channels, and **sm** for switching to mute:

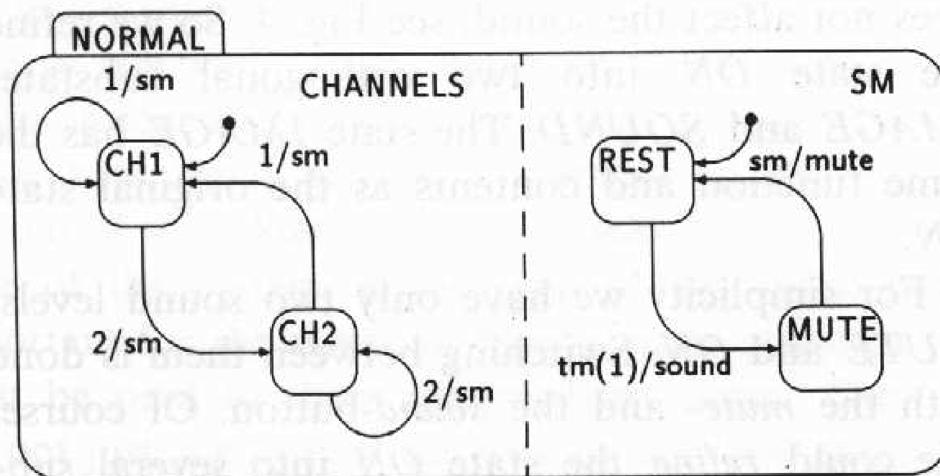


Fig. 5.

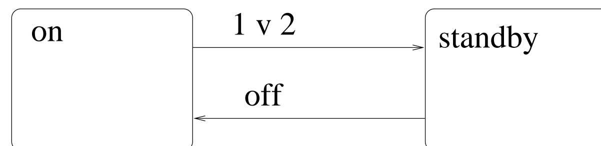
- When a channel button (1 or 2, for simplicity) is pressed, one switches to that channel and the internal event **sm** is generated. This causes the event **mute** by a transition in **sm** to state **mute**, and the sound will be turned off.
- After **one second** the event **sound** is generated to turn the sound on, again. This is done by the special **time-out** event **tm(1)**.

Thus one sees that orthogonal components can communicate by generating events which are broadcast, and that this can be done in a time-dependent manner: introducing the **generation of events** $e/a_1; \ldots ; a_n$ and **time-out events tm(1), ...**

In general the label of a transition consists of two parts: a **trigger** that determines if and when a transition will be taken, and an **action** that is performed when a transition is taken. This action is the generation of a **set of events**.

0.4.4 Fourth concept: Compound events

- When in state **standby**, dependent on whether one presses button 1 or 2 one makes sure to switch to states **ch1** or **ch2** in **on**. This is indicated as follows:



- In general one can label transitions by **compound events** s.a. $(\neg a \wedge b) \vee c, a \wedge b, c \vee d, \neg a$, etc
- E.g., in:

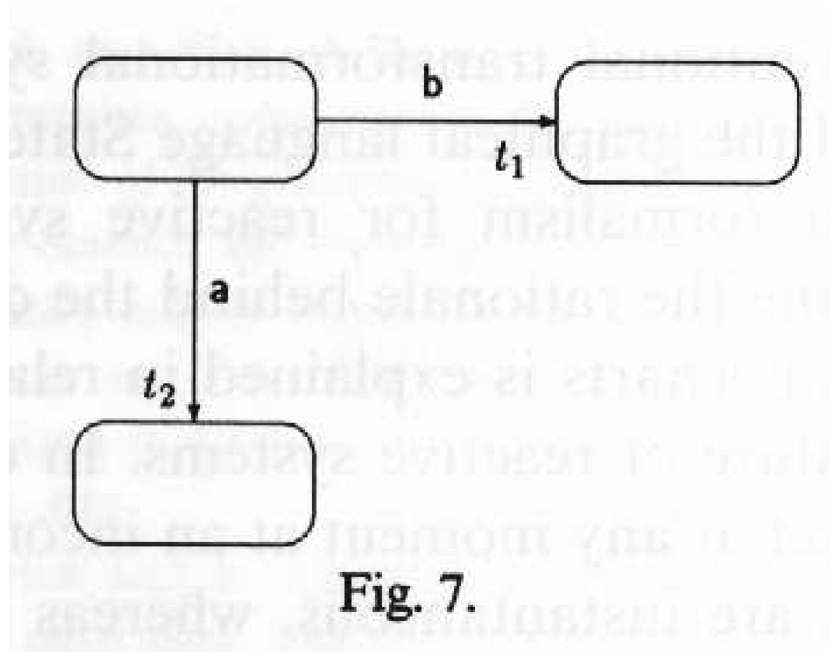


Fig. 7.

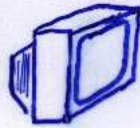
a can be replaced by $a \wedge \neg b$ to express priority of event b over event a.

In a nutshell, one may say with David Harel:

$$\text{Statecharts} = \text{Mealy Machines} + \text{depth}$$

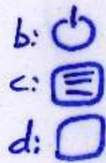
$$+ \text{orthogonality} + \text{broadcast} + \text{data}$$

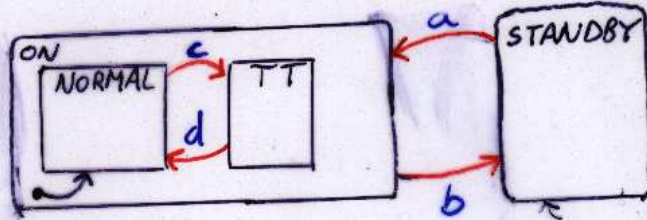
Extensions of the FSM



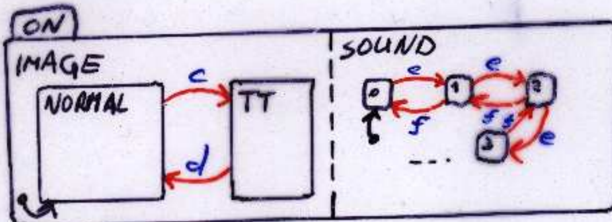
TV set + remote control
events: pressing buttons

- structured states & interupts

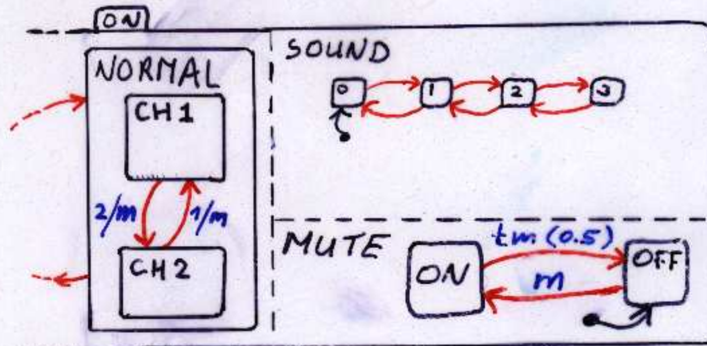




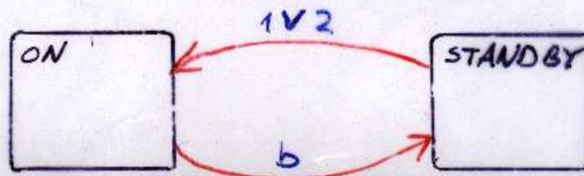
- parallelism (orthogonality)



- generation of events
  e/a₁;...;aₙ



- compound events
  (¬a ∧ b) ∨ c
  etc.
  a ∧ b



0.5 Time

- The elementary unit of observation in a reactive system is the **event**
- The **environment** sends events to the system to **trigger computations**, the system reacts to the environment by sending, or **generating**, events.
- Events are also means of **communication** between parts of a system.
- Because one wants to specify reactive systems at the **highest level of abstraction** in a discrete fashion, events are **discrete signals**, occurring at a **point in time**.

- Events have **no duration**; they are generated from one state to another. Hence, transitions have a discrete uninterruptable nature and **all time is spent in states**.

- **This has an important reason**:
  In a reason system new inputs may arrive at any moment. Therefore the current state it is in should be always clear. Since transitions have no duration, there are no “transient” periods in between states.
  Therefore, the reaction on a possible input is always well defined.

- Of course this is an abstraction from reality. (At deep levels of electronic implementations, one encounters levels where discrete reasoning makes no sense anymore)
  Statecharts is meant to be a high level specification language, where this abstraction can be maintained and is appropriate.

How long is the reaction time?

- We know that transitions have no duration, but **when** do they take place, relative to the trigger? And:

  HOW LONG DOES IT TAKE THE SYSTEM TO COMPUTE A REACTION UPON AN EXTERNAL EVENT?

- For transformational systems this is easy — the only important distinction is between finite and infinite values (corresponding to a final state or no final state)
- For reactive systems this is not enough:

  We have to know when an output occurs relative to the events in the input sequence (see Brock-Ackermann paradox)

  $\Longrightarrow$

  One has to determine **what the reaction time of a sequence is**.

What's the reaction time of a reactive systems upon an external event in the high level Specification Language Statecharts?

Possibility 1 : Specify a concrete amount of time for each situation. This forces us to quantify time right from the beginning. **Clumsy**, and not appropriate at this stage of specification where one is only interested in the relative order and coincidence of events.

Possibility 2 : Fix reaction time between trigger **a** and corresponding action **a** within **e/a** (the label of a transition) upon 1 time unit.

Doesn't work: Upon refining **question/answer** to a **question/consult** and a **consult/answer** transition, there's a change of time, which may have far reaching effects (because of **tm(n)**-events, e.g.)

$\Longrightarrow$

A fixed execution time for syntactic entities (transitions, statements, etc.) is not **flexible** enough.

Possibility 3 : Leave things open: say only that execution of a reaction takes some positive amount of time, and see at a later stage (closer to the actual implementation) how much time things take.

Clumsy, introduces far too much nondeterminism.

Reaction time of a system (2)

Summary : We want the execution time associated to reactions to have following properties:

- It should be accurate, but not depending on the actual implementation.
- It should be as short as possible, to avoid artificial delays.
- It should be abstract in the sense that the timing behavior must be orthogonal to the functional behavior.

Only choice that meets all wishes is **zero reaction time**.

As a result all objections raised w.r.t. the possibilities mentioned on the previous page are met!

- Now, for instance, upon refining transition **question/answer** from previous page into two transitions, the reaction time of this refinement is the same as that of the original transition.
- Objection 3 on the previous transparency is resolved, too.
- Finally, also objection 1 (on previous transparency) is met, because $0 + 0 = 0$!

This choice, that the reaction time between a trigger and its event is zero, is called **Berry's synchrony hypothesis**.

Is this implementable? No, a real computation takes time. But in actual implementation this means:

The reaction comes before the next input arrives,

or, so to say,

Reactions are not infinitely fast but fast enough.

See the following figure:

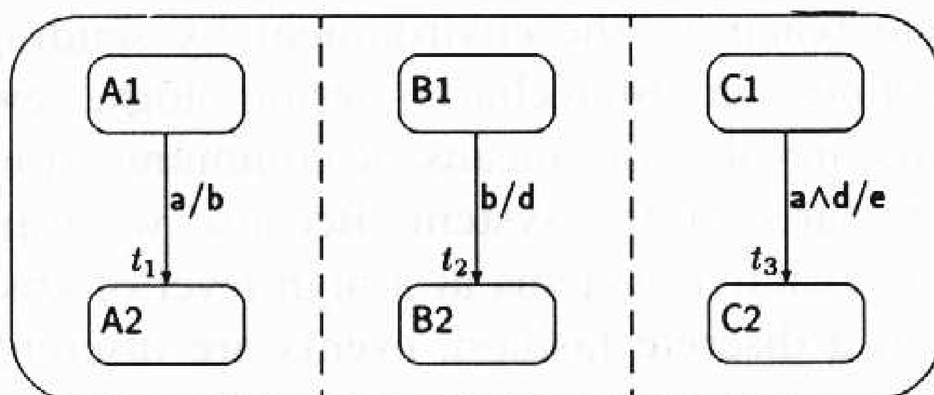


Fig. 6.

A consequence is that transition t_3 is taken!!

Negations and paradoxes

- Idea of immediate reaction works fine as long as transitions only triggered by primitive events, or or conjunctions and disjunctions of them.
- However, one also needs **negations of events** to trigger a transition. E.g.: to specify priority:

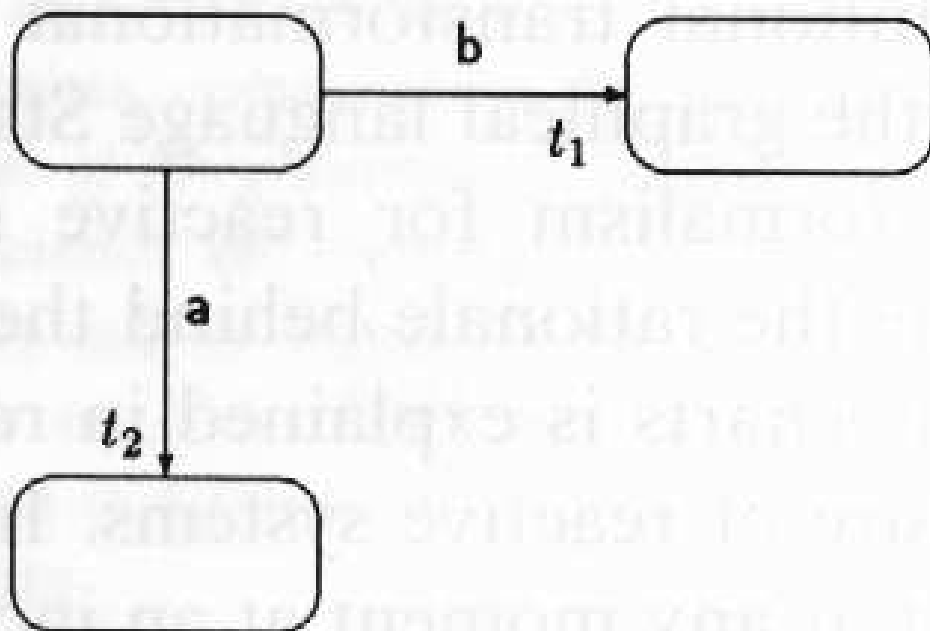


Fig. 7.

- Problem: What semantics to give to Statecharts in the next figure?

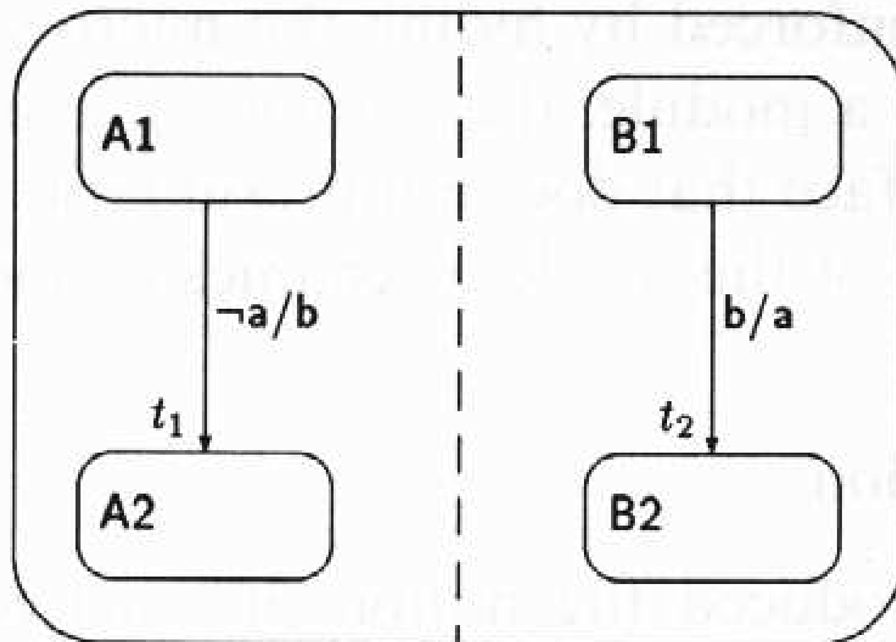


Fig. 8.

If **a**is absent, i.e., ¬ **a** holds as condition, transition t_1 is taken, i.e., **b** is generated, and hence t_2, i.e., **b/a** is taken, generating **a within the same time unit**, i.e., in zero time, hence transition t_1 should not be taken.

But that means that event **b** is not generated, and hence event **a** is not generated, so transition t_1 should be taken, etc.

$\Longrightarrow$ PARADOXON!

Negations and paradoxes (2)

Solution : Introduce two levels of time

- **Macro steps**, for counting time, (these are observable) time steps, and
- **Micro steps**, which describe the causal chain within reactions.
  Every macro-step is then divided in an arbitrary but finite number of micro-steps.

This sequence of micro-steps has only an operational meaning.

Alternative approaches

- Problem with introducing macro-steps only within our formalism (and no micro-steps) is that the semantics is no longer **globally consistent** in case of the following statechart:

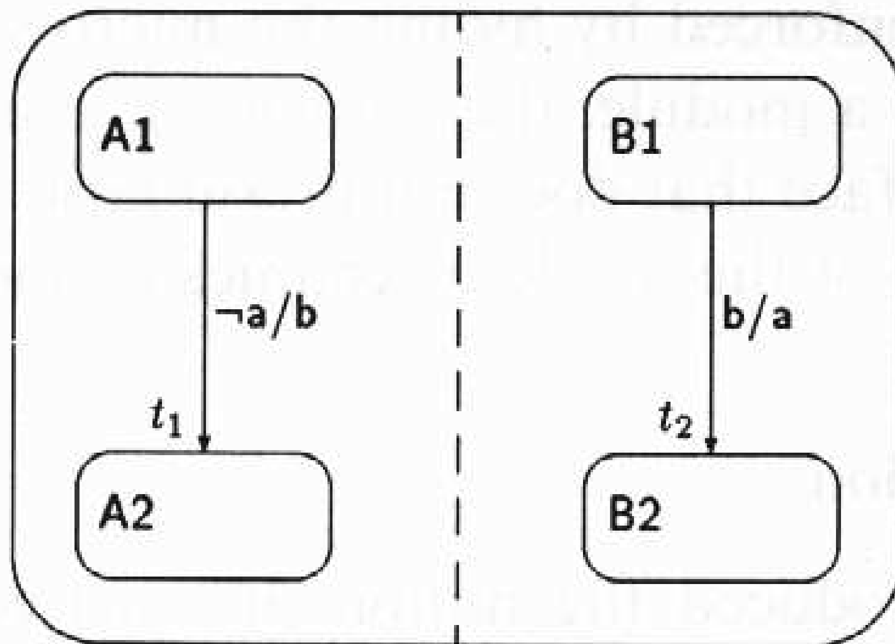


Fig. 8.

- Another problem is: What semantics to give to the statechart in the following figure?

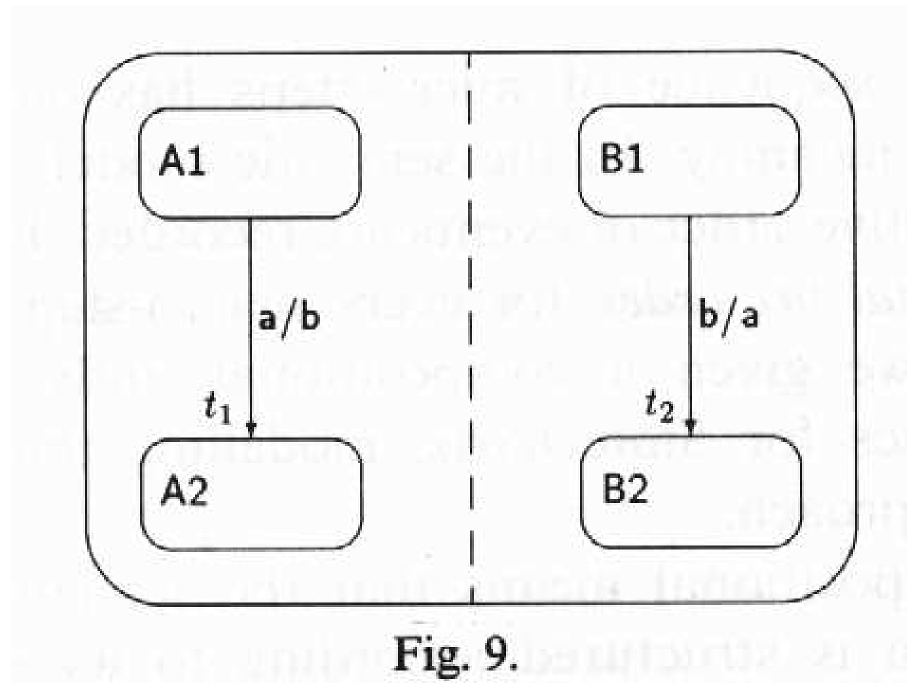


Fig. 9.

Causality dictates that no such transitions are "triggered themselves": there should be a causally acceptable chain of reaction steps.

- **Responsiveness**: reactions are simultaneous with their triggers.

It can be proved that no semantics can be both globally consistent, causal, and responsive.

E.g., in the synchronous language ESTEREL, programs are disallowed which violate causality. The compiler detects these, and refuses to compile them.

Conclusion

- Reactive systems are fundamentally different from transformational systems
- We explained the design decisions behind Statecharts in relation to the specific nature of reactive systems
- Time is passed in states, transitions are instantaneous
- To avoid accumulation of time in reactive chains, the reaction time should be zero
- We pointed out that several semantic problems arise when reaction time is zero, and how to circumvent these problems.

  For solutions the reader is referred to the literature, see the paper by Huizing and Hooman, and the book by David Harel and Michal Politi:

  > David Harel and Michal Politi. Modeling Reactive Systems with Statecharts. McGraw-Hill, 1998.

Session II

Introduction to Statemate

Abstract

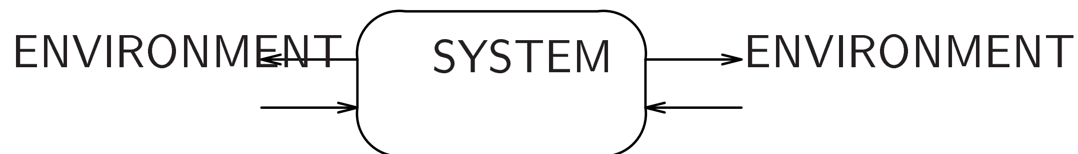
The notion of a reactive system and the language Statecharts were introduced in the last session. We explained the rationale behind the design decisions of Statecharts in relation to the specific nature of reactive systems.

This time, the role of models in a system development life cycle is described. We introduce three languages to characterize reactive systems from different views. This yields to a brief description of the STATEMATE toolset.

Literature: David Harel and Michal Politi. Modeling Reactive Systems with Statecharts. McGraw-Hill, 1998.

Last Session

We wanted to describe reactive systems:



Solution

$$\text{Statecharts} = \text{Mealy Machines} + \text{depth}$$

$$+ \text{orthogonality} + \text{broadcast} + \text{data}$$

0.6 Specification in a systems life cycle

- Identify several phases in the development life cycle of a system
- **Classic waterfall model**: requirements analysis, specification, design, implementation, testing, and maintenance.
- Other approaches center around prototyping, incremental development, reusable software, or automated synthesis.
- Most proposals contain a requirements analysis phase. Specification errors and misconceptions should be discovered in that early phase.
- Correcting errors in later stages is extremely expensive.
- Special languages are therefore used in the requirements analysis phase to specify a model of the system, and special techniques are used to analyze it extensively.

System's life cycle

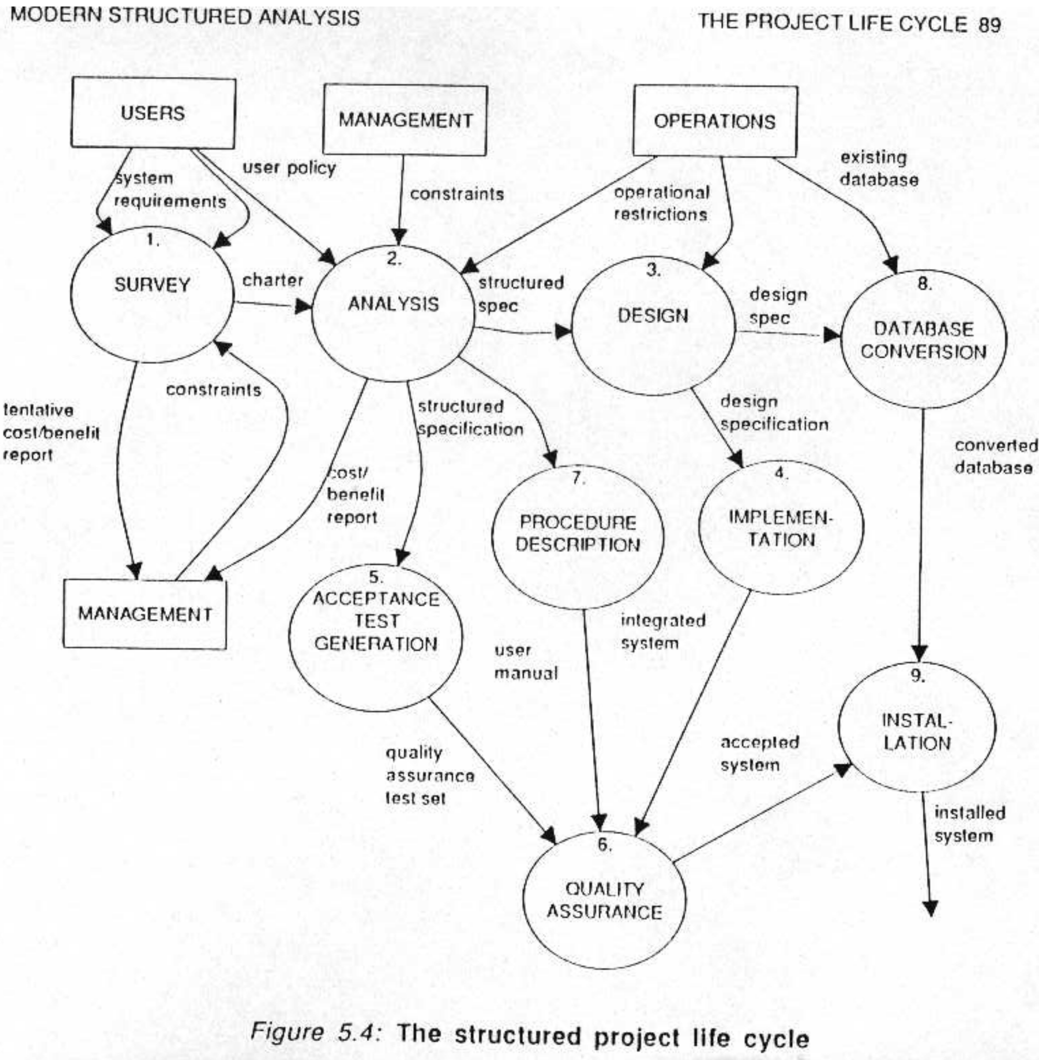


Figure 5.4: **The structured project life cycle**

System model

- A good model is important for all participants in the system's development.

- Having a clear and executable model the functionality and behavior can be approved before investigating heavily in the implementation stages.

- The specification team uses modeling as the main medium for expressing ideas.

0.7 Methodology

A methodology provides guidelines for performing the processes that comprise the various phases.

Concentrating on the modeling and analysis phase, a methodology consists of the following components:

- The methodology's underlying approach and the concepts it uses.

- The notation used, that is, the modeling languages with their syntax and semantics.

- The process prescribed by the methodology, that is, which activities have to be carried out to apply the methodology and in what order.

- The computerized tools that can be used to help in the process.

Here, we focus on notation and describe the STATEMATE toolset.

0.8 Reactive systems

The Statecharts language is especially effective for reactive systems.

A typical reactive system exhibits the following distinctive characteristics:

- It continuously interacts with its environment, using inputs and outputs that are either continuous in time or discrete.
- It must be able to respond to interrupts, i.e., high-priority events.
- Its operation and reaction often reflect stringent time requirements.
- It is very often based on interacting processes that operate in parallel.

Examples

On-line interactive systems : e.g., automatic teller machines, flight reservation systems

Computer-embedded systems :avionics, automotive, and telecommunication systems

Control systems : such as chemical and manufacturing systems.

Example: The early warning system A system model constitutes a tangible representation of the system's conceptual and physical properties and serves as a vehicle for the specifier and designer to capture their thoughts.

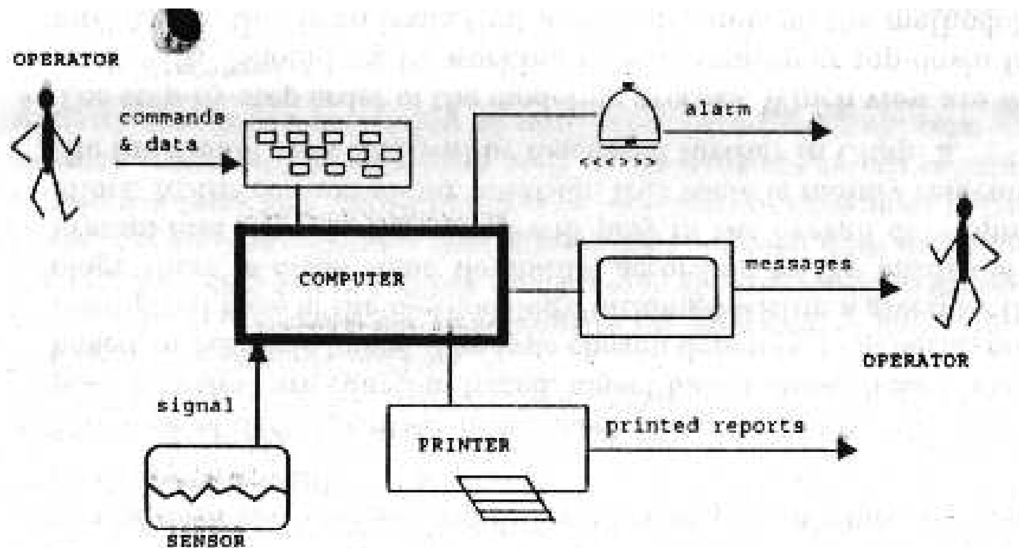


Figure 1.1 The early warning system (EWS).

0.9 Characteristics of models

Beside for communication, systems models should also be used for inspection and analysis.

When the model reflects some preexisting descriptions, such as requirements written in natural language, it is useful to keep track of how the components of the developing model are derived from the earlier descriptions.

The modeling languages used in STATEMATE have been designed with several important properties in mind:

- to be intuitive and clear
- to be precise
- to be comprehensive
- to be fully executable

How to achieve these properties?

- To achieve clarity, elements of the model are represented graphically wherever possible.
- For precision, all languages features have rigorous mathematical semantics
- Comprehension comes from the fact that the languages have the full expressive power needed to model all relevant issues, including the what, the when, and the how.
- For executability, the behavioral semantics is detailed and rigorous enough to enable the model to be executed (or be used to generate code).

0.10 Modeling Views

Building a model can be considered as a transition from ideas and informal descriptions to concrete descriptions that use concepts and predefined terminology.

Here, the descriptions used to capture the system specification are organized into three views: the **functional**, the **behavioral**, and the **structural**

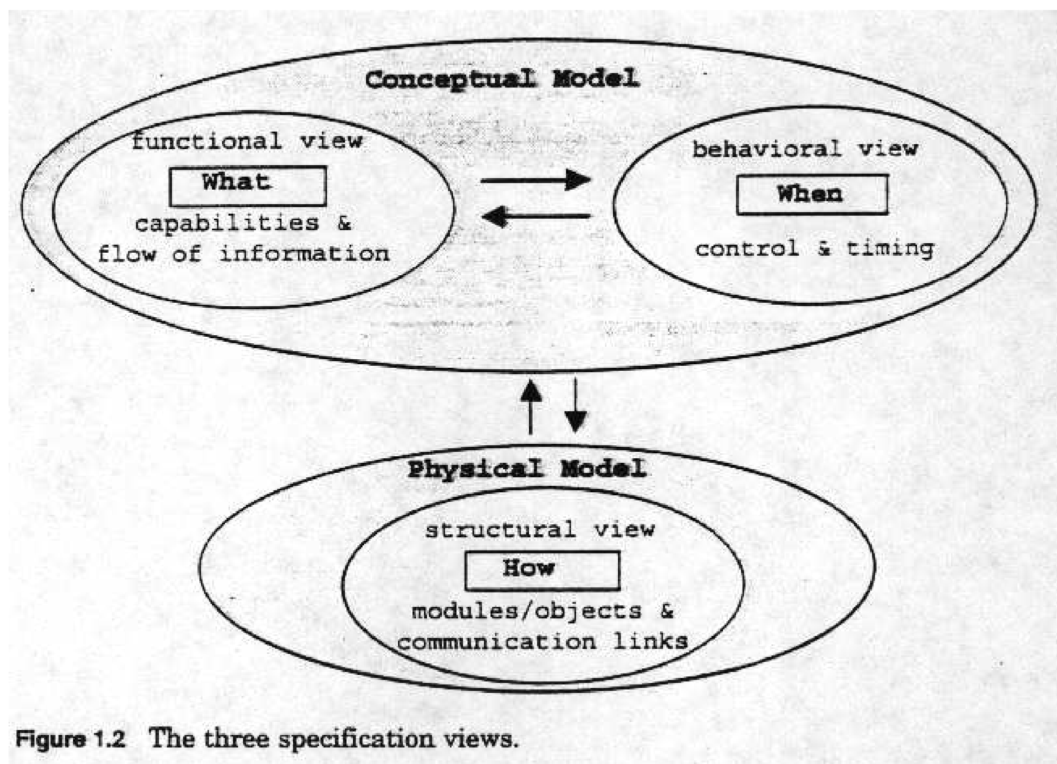


Figure 1.2 The three specification views.

The three views

Functional view : The functional view captures the “what”. It describes the system’s functions, processes, or objects, also called activities, thus pinning down its capabilities. This view includes the inputs and outputs of the activities.

Behavioral view : The behavioral view captures the “when”. It describes the system’s behavior over time, including the dynamics of activities, their control and timing behavior, the states and modes of the system, and the conditions and events that cause modes to change and other occurrences to take place.

Structural view : The structural view captures the “how”. It describes the subsystems, modules, or objects constituting the real system and the communication between them.

While the two former views provide the conceptual model of the system, the structural view is considered to be its physical model.

The main connection between the conceptual and physical models is captured by specifying the modules of the structural view that are responsible for implementing the activities in the functional view.

Modeling heuristics

Modeling heuristics are guidelines for how the notation should be used to model the system.

- The mapping between the methodology's concepts and the elements allowed in the notation.
- The type of decomposition to be used: e.g., function based, object based, mode based, module based, or scenario based.
- The step-by-step order of the modeling process: bottom-up or top-down

0.11 The Modeling Languages

The three views of a system model are described in our approach using three graphical languages.

- **Activity-charts** for the functional view,
- **Statecharts** for the behavioral view,
- and **Module-charts** for the structural view.
- Additional non-graphical information related to the views themselves and their inter-connections is provided in a **Data Dictionary**

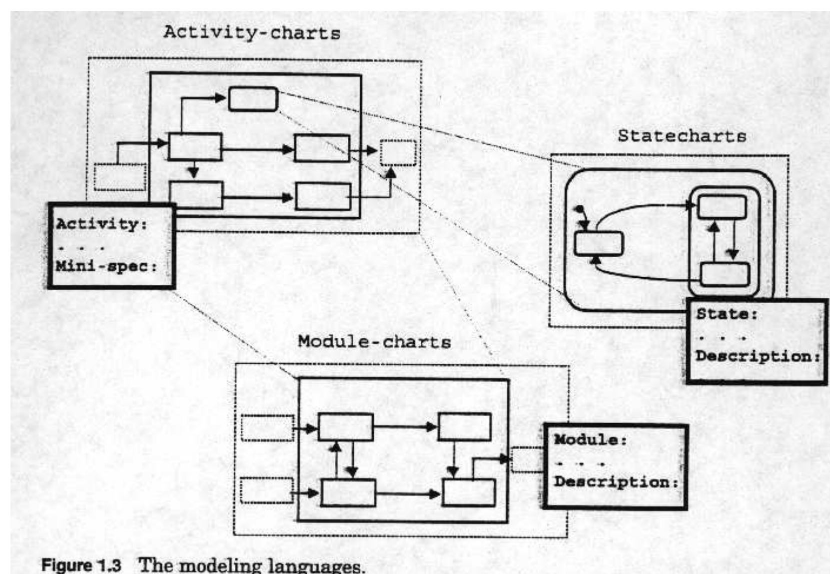


Figure 1.3 The modeling languages.

0.11.1 Activity-charts

Activity-charts can be viewed as multilevel data-flow diagrams. They capture functions, or activities, as well as data-stores, all organized into hierarchies and connected via the information that flows between them.

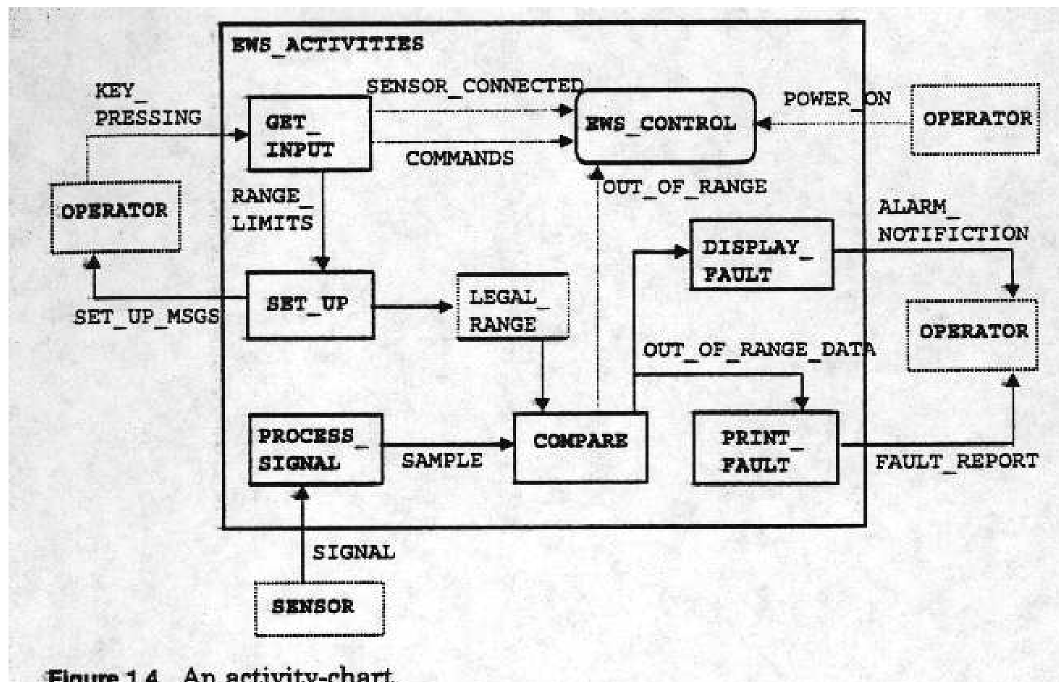


Figure 1.4 An activity-chart.

0.11.2 Non-graphical information

In addition to the graphical information, each element in the described has an entry in the Data Dictionary, which may contain non-graphical information about the element.

For example, the activity entry contains fields called mini-spec and long description, in which it is possible to provide formal and informal textual descriptions of the activities workings.

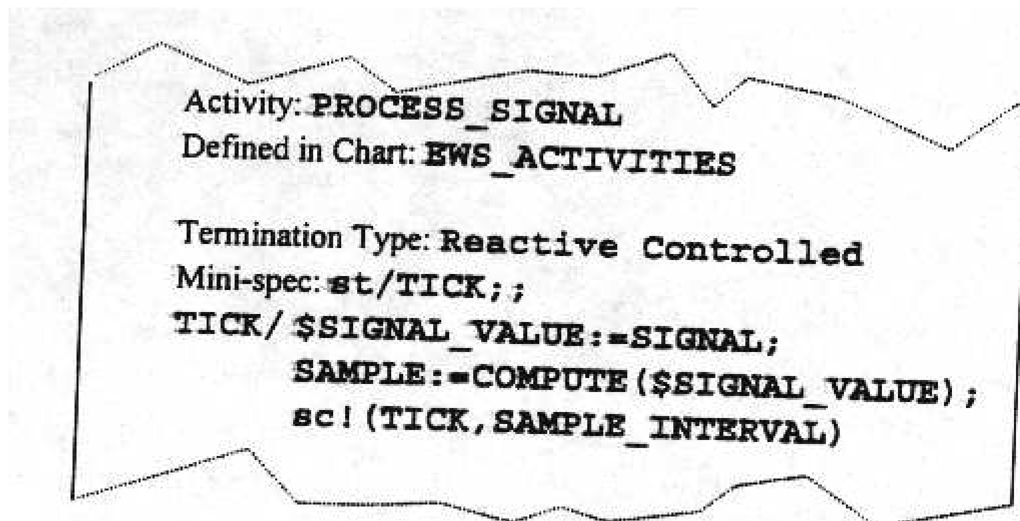


Figure 1.5 An activity entry in the Data Dictionary.

0.11.3 Statecharts

Statecharts constitute an extensive generalization of state-transition diagrams. They allow for multilevel states decomposed in an and/or fashion, and thus support economical specification of concurrency and encapsulation. They incorporate a broadcast communication mechanism, timeout and delay operators for specifying synchronization and timing information, and a means for specifying transitions that depend on the history of the system's behavior.

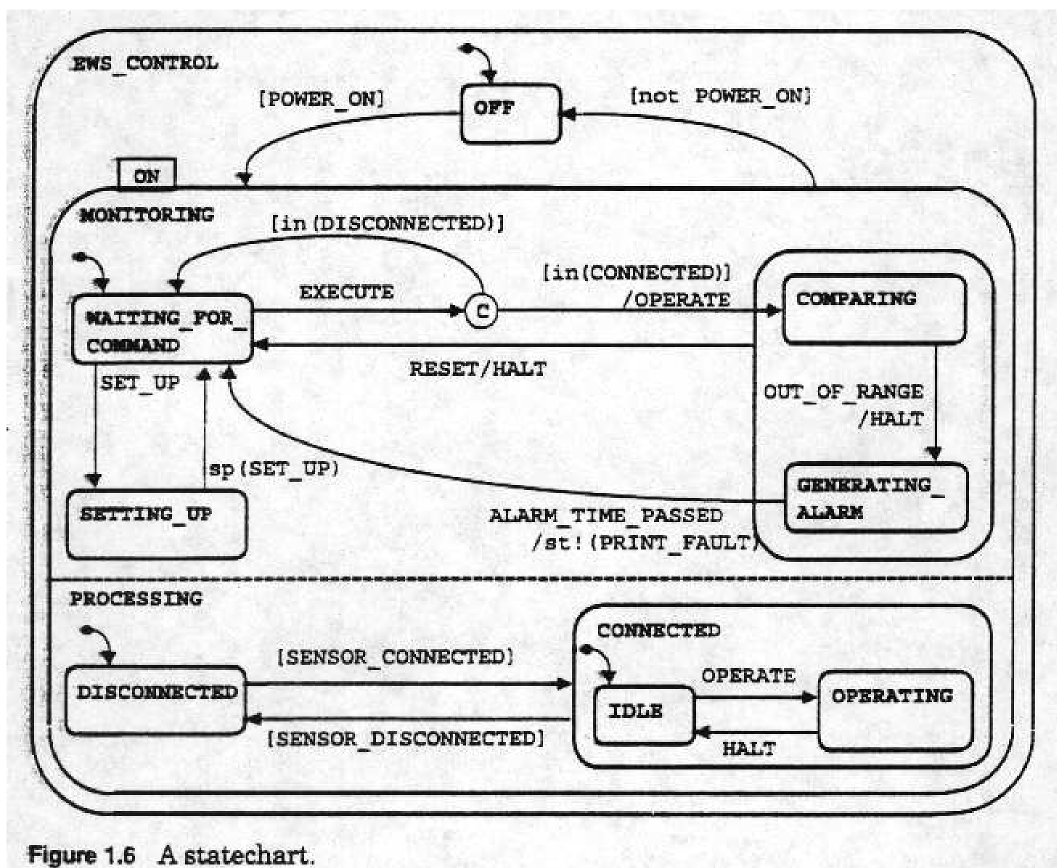


Figure 1.6 A statechart.

0.11.4 Module-chart

A module-chart can also be regarded as a certain kind of data-flow diagram or block diagram. Module-charts are used to describe the modules that constitute the implementation of the system, its division into hardware and software blocks and their inner components, and the communication between them.

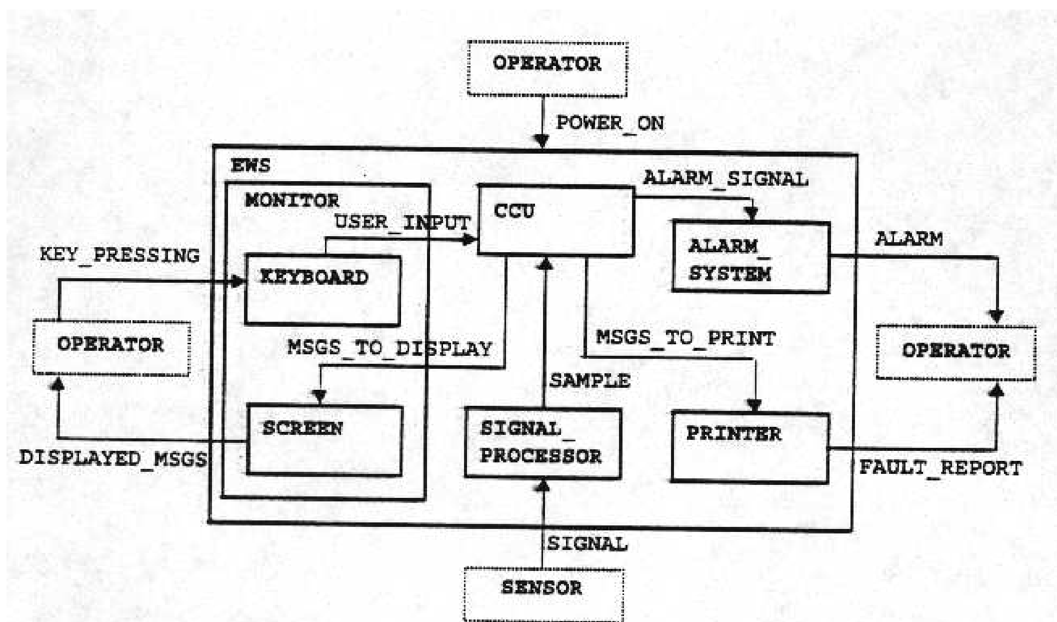
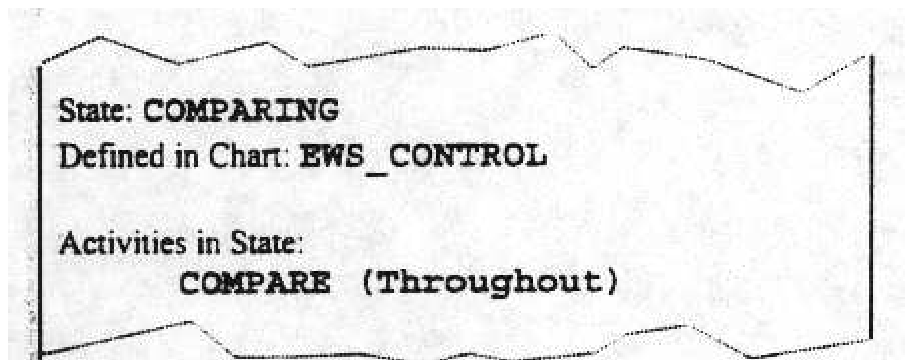


Figure 1.7 A module-chart.

0.12 Relationship between the languages

The relationship between the concepts of the three views are reflected in corresponding connections between the three modeling languages.

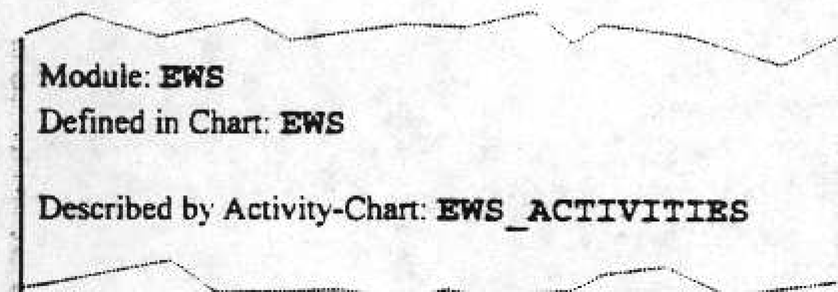
Most of these connections are provided in the Data Dictionary, and they tie the pieces together, thus yielding a complete model of the system under development.



State: COMPARING
Defined in Chart: EWS_CONTROL

Activities in State:
COMPARE (Throughout)

Figure 1.8 Specifying an activity throughout a state.



Module: EWS
Defined in Chart: EWS

Described by Activity-Chart: EWS_ACTIVITIES

Figure 1.9 An activity-chart describing a module.

Handling large-scale systems

- The languages allow to split large hierarchical charts into separate ones:

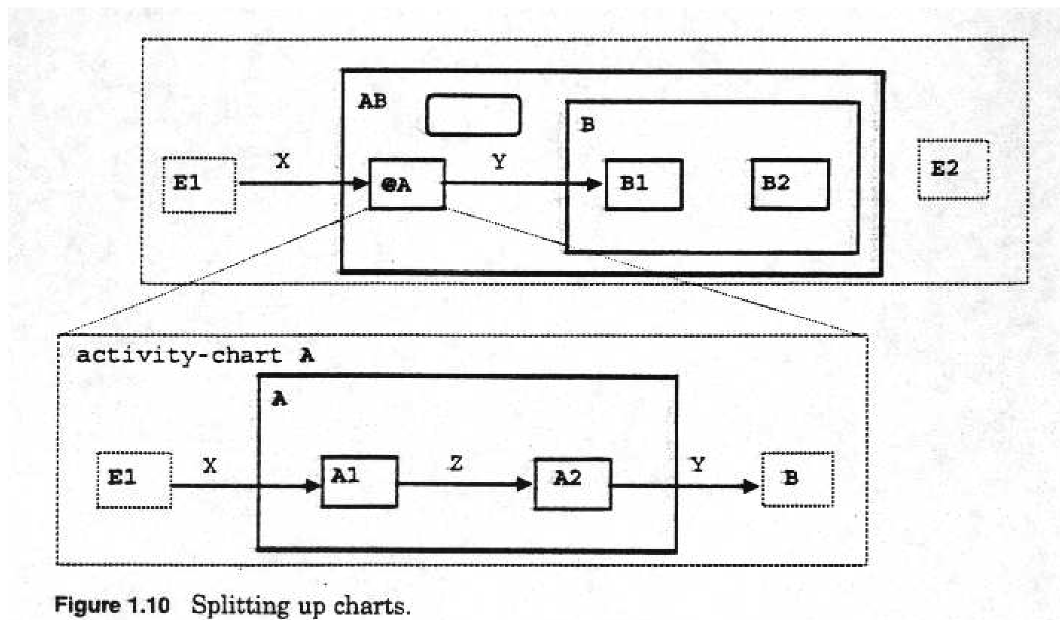


Figure 1.10 Splitting up charts.

- Also, coping with visibility and information hiding by setting scoping rules of elements in the model.

- Moreover, **generic charts** and **user-defined types**.

The STATEMATE toolset

STATEMATE has been constructed to "understand" the model and its dynamics. The user can then execute the specification by emulating the environment of the system under development and letting the model make dynamic progress in response.

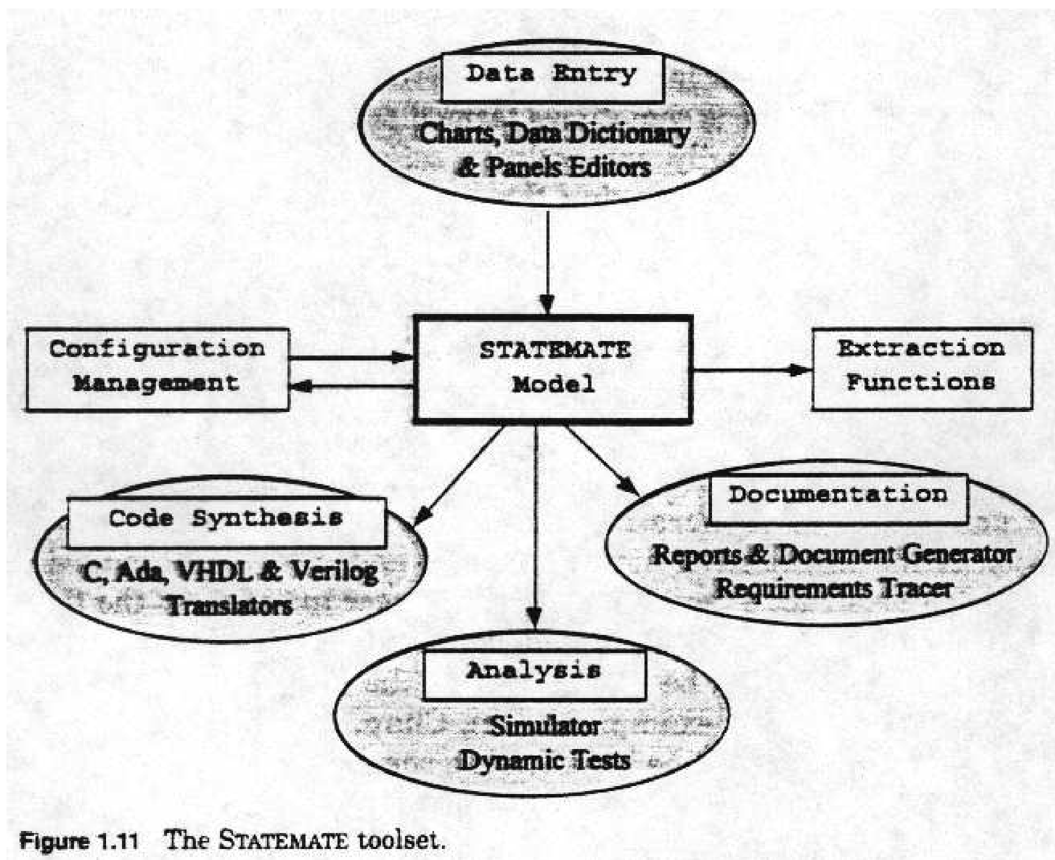
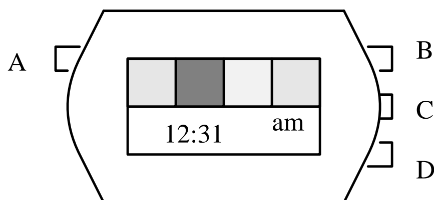


Figure 1.11 The STATEMATE toolset.

0.13 A larger example: Alarm watch

As an example of a statechart we use that of a simple digital watch with four buttons A, B, C and D like in the below picture:



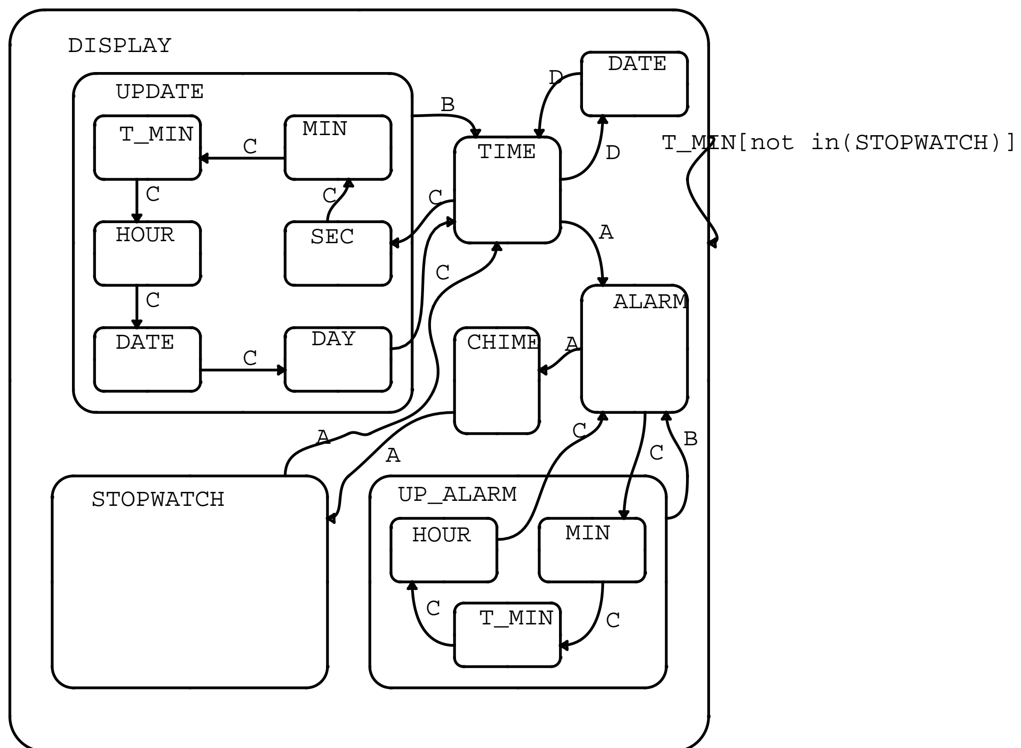
The watch can show the time or date, has the possibility to beep every hour, an alarm, a stopwatch, a light and an indication that the battery must be replaced.

The following events are considered as external:

- A, B, C and D describe the pushing of the four buttons and B_up the release of button B.
- The events Bt_In, Bt_Rm, Bt_Dy and Bt_Wk describe respectively the putting in, removal, drop dead and weakening of the battery.
- T_hits_Hr describes that the internal time has reached a whole hour and T_Hits_Tm describes that the internal time has reached the alarm time.
- T_Min describes that there are two minutes passed since for the last time a button has been pushed.

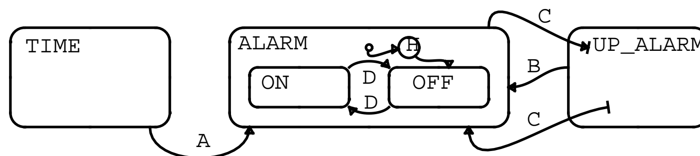
Display control

The below statechart shows the $display$ state of the watch. There is a special state $up - alarm$ for the changing of the internal state of the alarm. Note that T_min takes care of the resetting of a state, except the $stopwatch$, to the default state $time$ if nothing has happen since the last two minutes.

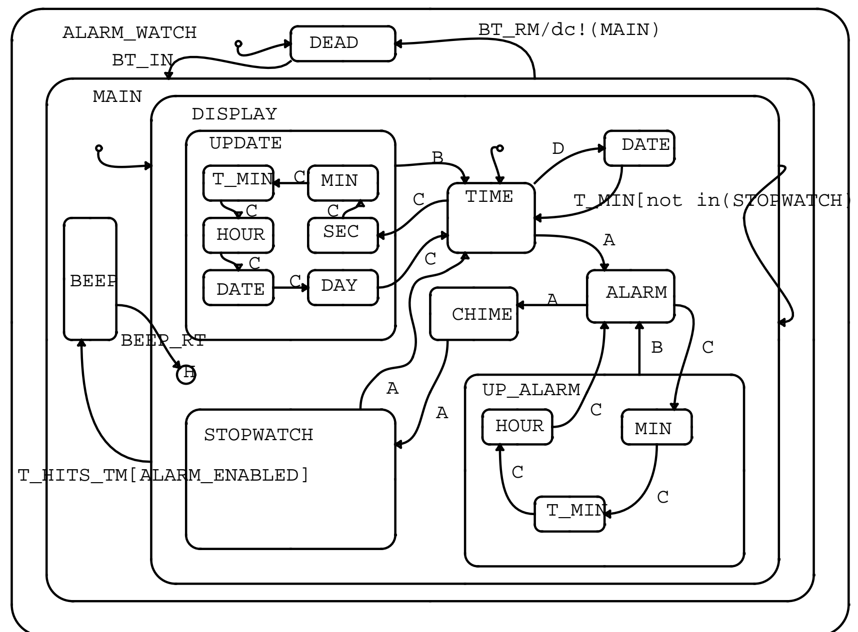


0.13.1 History of states

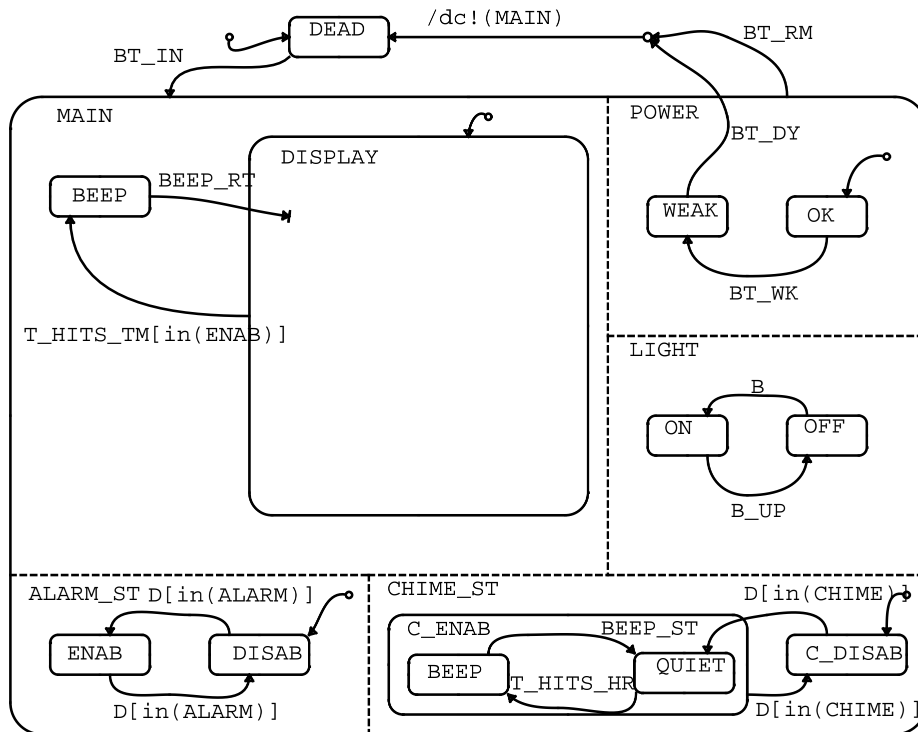
A frequently used way to enter a group of states is by the history of that group. The most simple example of this is the one where you enter the most recently visited state of a group. In the watch example this happens in the zoom-in of the $alarm$ state with two substates on and off. The problem is that the initial default is the off state but when we put on the alarm we want to get back the next time in state on. In the next statechart this described by the **H** connector.



The following statecharts sums up the till now developed watch. It contains a $beep$ state that is entered when event T_Hits_Tm occurs, provided the alarm is on (the condition $alarm_enabled$ shall be explained later), and is left when event $Beep_Rt$ occurs. This event $Beep_Rt$ is an abbreviation of $(A\ or\ B\ or\ C\ or\ D\ or\ tm(en(beep), 30))$ wherein tm stands for 'timeout' and en for 'entered'. This means that the watch returns to the previous state (because of $\mathbf{H}^*$) when one of the four buttons is pushed or 30 seconds are passed since the entering of $beep$.

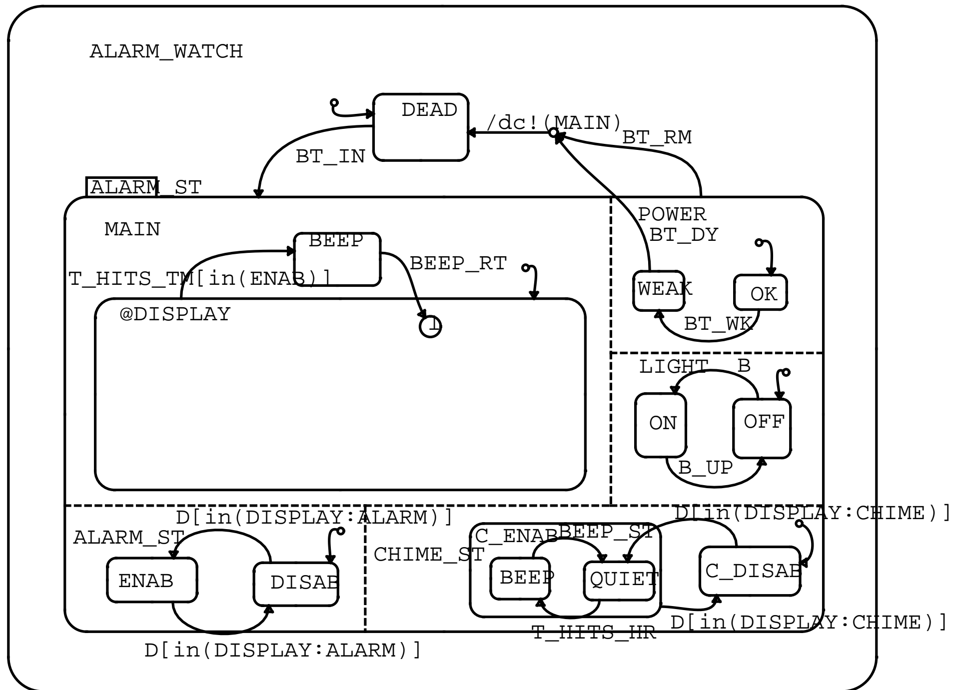


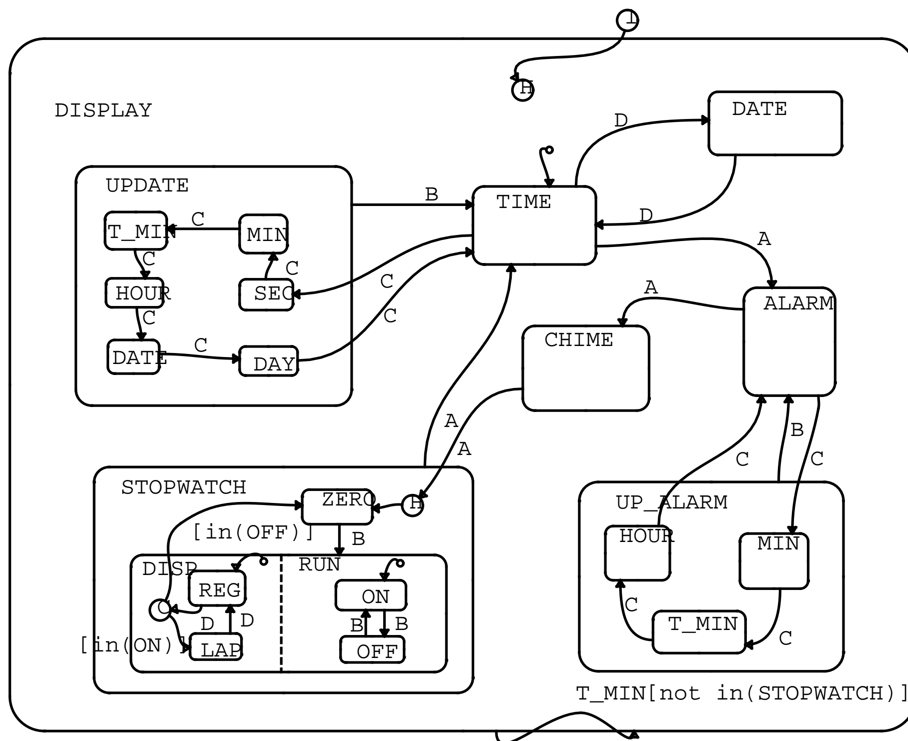
The watch contains also orthogonal states on different levels. On the higher levels there are besides the state $main$ of the above statechart also four other and-components as is illustrated in the next statechart:



The event $Beep_st$ is an abbreviation for $tm(en(c_enab.beep), 2)$ what means that the beeping stops after 2 seconds.

The complete statechart for the watch is:




DISPLAY
H
DATE
D
UPDATE
B
TIME
T_MIN
C
MIN
D
A
C
C
HOUR
SEC
C
A
ALARM
C
CHIME
C
DATE
C
DAY
A
B
A
C
C
STOPWATCH
ZERO
H
UP_ALARM
[in(OFF)]
B
HOUR
MIN
DISP
RUN
REG
ON
C
T_MIN
D
D
B
B
C
[in(ON)]
LAP
OFF
T_MIN[not in(STOPWATCH)]

We finish the description of statecharts with an overview of the syntax for events, conditions and actions. The general form for a label of a transition is $E[C]/A$ with E an event, C a condition and A an action.

0.13.2 Events

The basic events and condition are external, for example for the watch the pressing of a button is an external event and T_Hits_Tm is an external condition. Besides the external events the following internal events are allowed:

$entered(S)$,	abbreviation	$en(S)$,
$exit(S)$,	abbreviation	$ex(S)$,
$timeout(E, X)$,	abbreviation	$tm(E, X)$,
$true(C)$,	abbreviation	$tr(C)$,
$false(C)$,	abbreviation	$fs(C)$.

0.13.3 Actions

An action can be an uninterpreted event symbol, called primitive event, and causes then other transitions in the statechart. Furthermore actions can turn on or off uninterpreted condition symbols. The following primitive actions are allowed:

$make_true(C)$,	abbreviation	$tr!(C)$,
$make_false(C)$,	abbreviation	$fs!(C)$,
$history_clear(S)$,	abbreviation	$hc!(S)$,
$deep_clear(S)$,	abbreviation	$dc!(S)$.

Session III

Activity-Charts

Literature: Chapters 2 and 3 of "Modeling Reactive Systems with Statecharts", by David Harel and Michal Politi. McGraw-Hill, 1998.

1 Describing the functional view of a system

- Activity-charts are used to depict the functional view of a system under development (SUD), "what the SUD does".
- This view of a system is specified by
  - a hierarchy of functional components, called **activities**,
  - what kind of information is exchanged between these activities and is manipulated by them,
  - how this information flows,
  - how information is stored, and
  - how activities are **started and terminated**, i.e., **controlled**, if necessary, and whether activities are **continuous**, or whether they **stop by themselves**.
- Activity-charts are kind of **hierarchical data flow diagrams**:

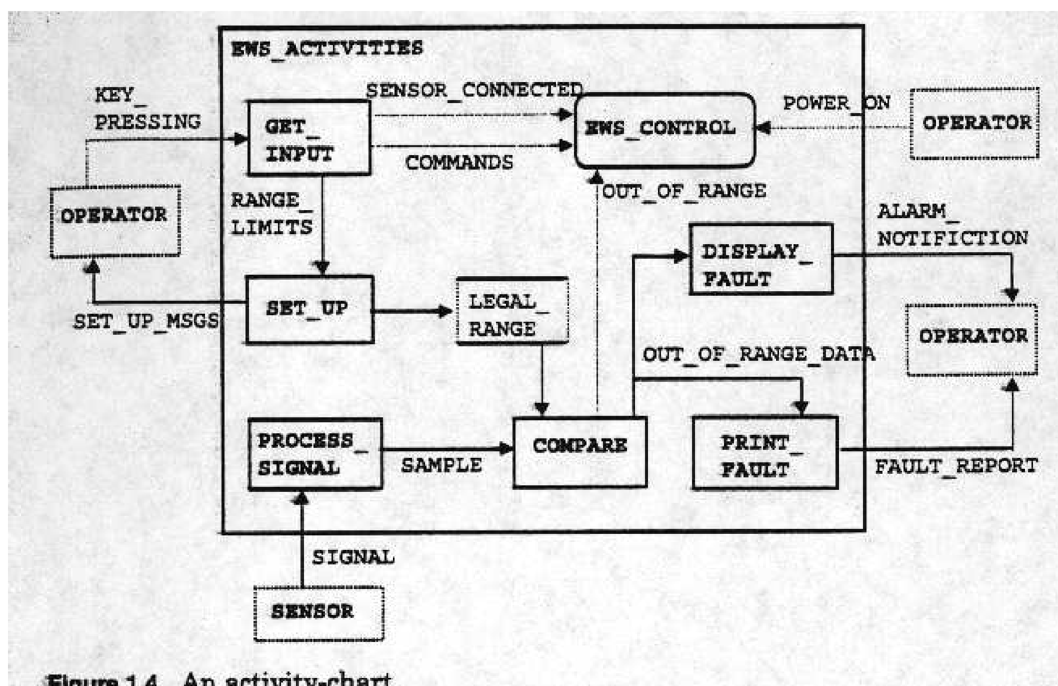


Figure 1.4 An activity-chart.

1.1 Functional decomposition of a System

The functional view of a system specifies the system's **capabilities**.

- It does so in the context of the system's environment, that is, it defines the environment with which the system interacts and the interface between the two:

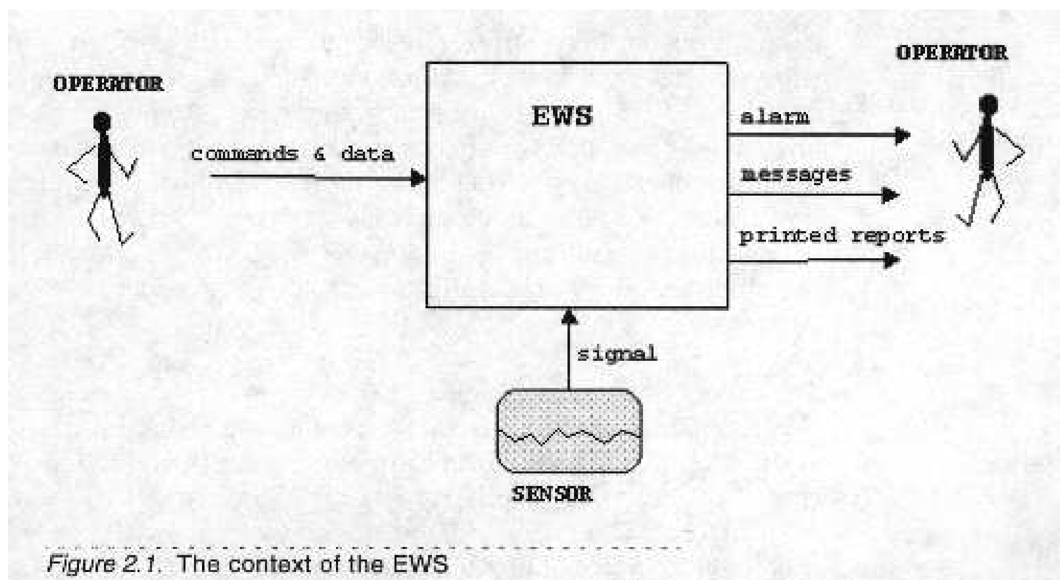


Figure 2.1. The context of the EWS

- This functional view does not address the physical and implementation aspects of the system; the latter is done in its **structural** view, i.e., its **module-chart**:

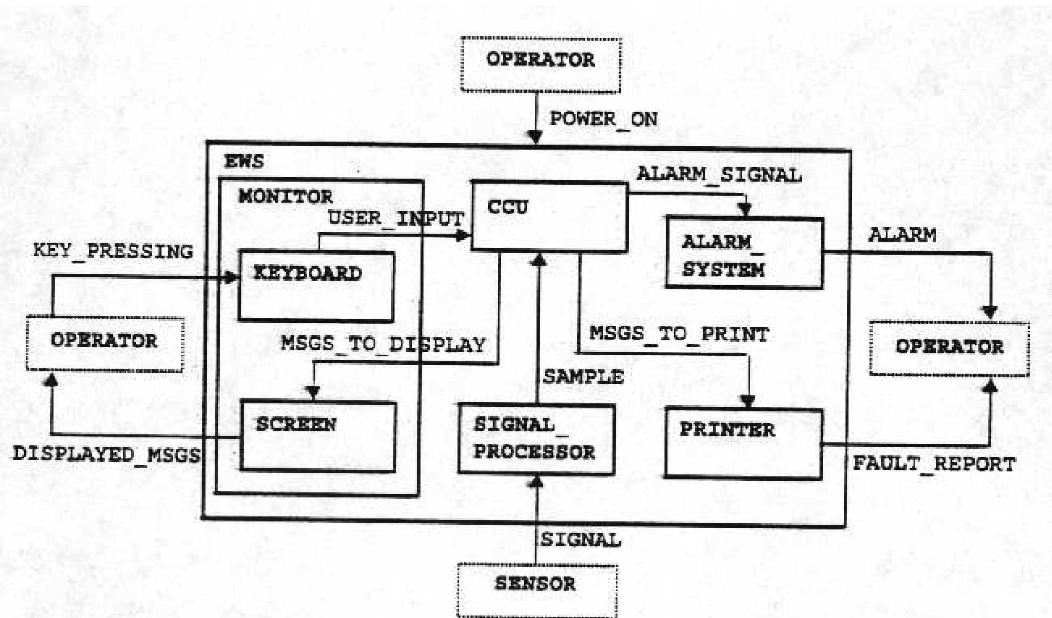


Figure 1.7 A module-chart.

- Moreover it separates the dynamics and behavioral aspects of the SUD from its functional description. The former is done by its **behavioral** view, in its controlling Statecharts:

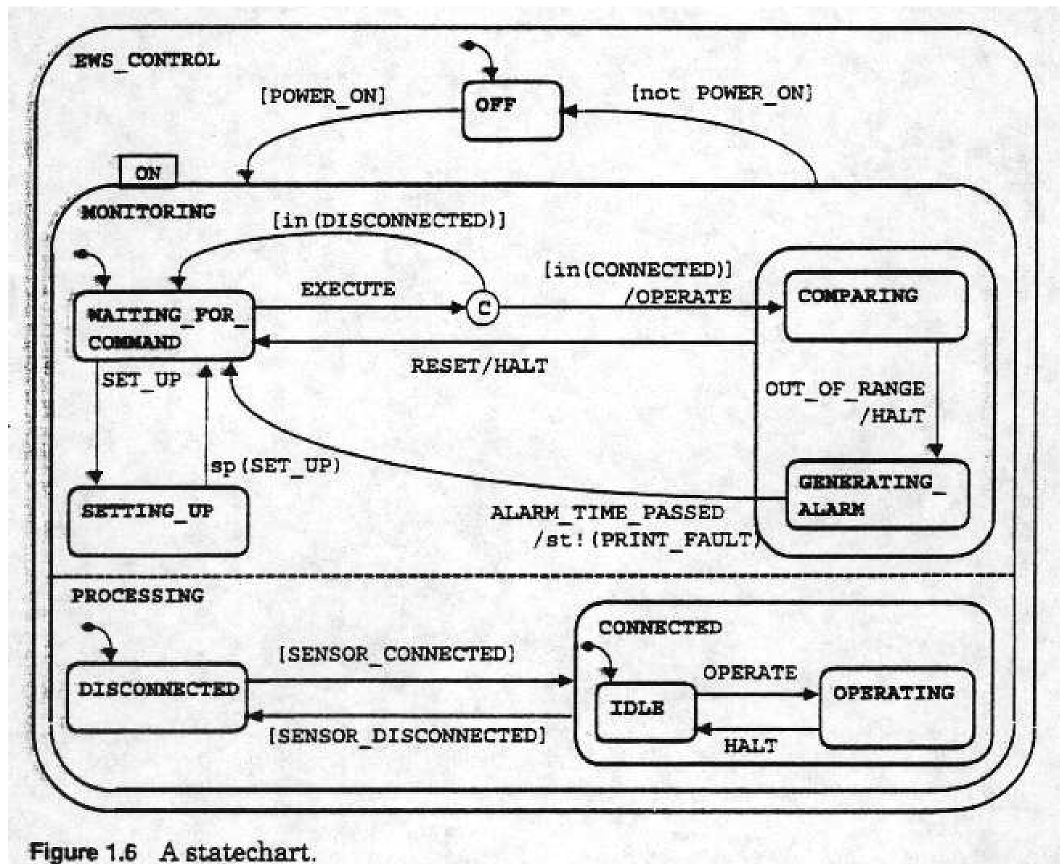


Figure 1.6 A statechart.

Example

The **functional view** tells whether a medical diagnosis system can monitor a patient's functions, and, if so, where it gets its input data and which functions have access to the output data.

The **behavioral view** tells under which conditions monitoring is started, whether it can be carried out parallel to temperature monitoring, and how the flow of control of the process of monitoring develops.

The **structural view** deals with the sensors, processors, monitors, software modules and hardware necessary to implement the monitoring system

The three views

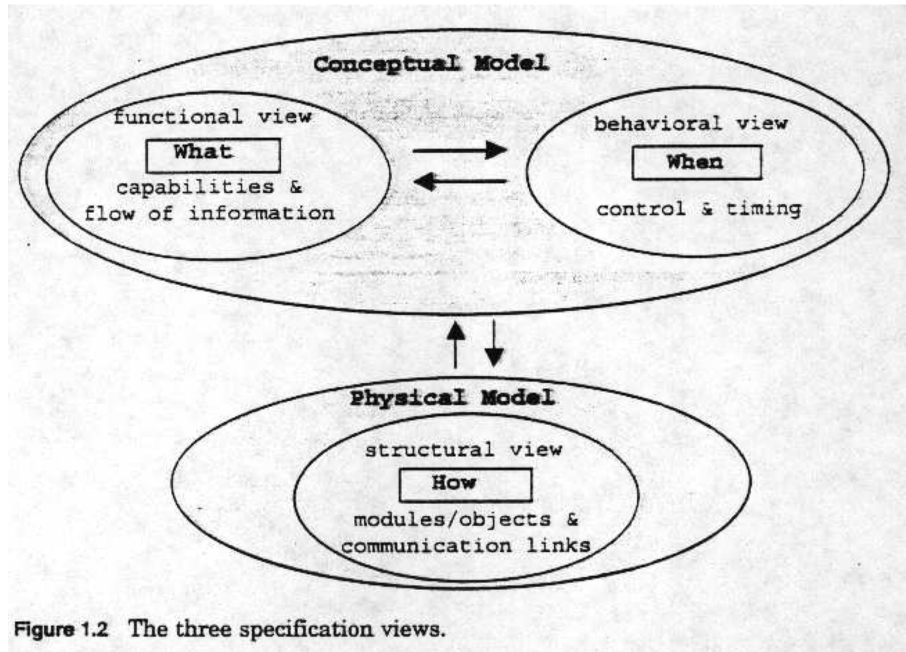


Figure 1.2 The three specification views.

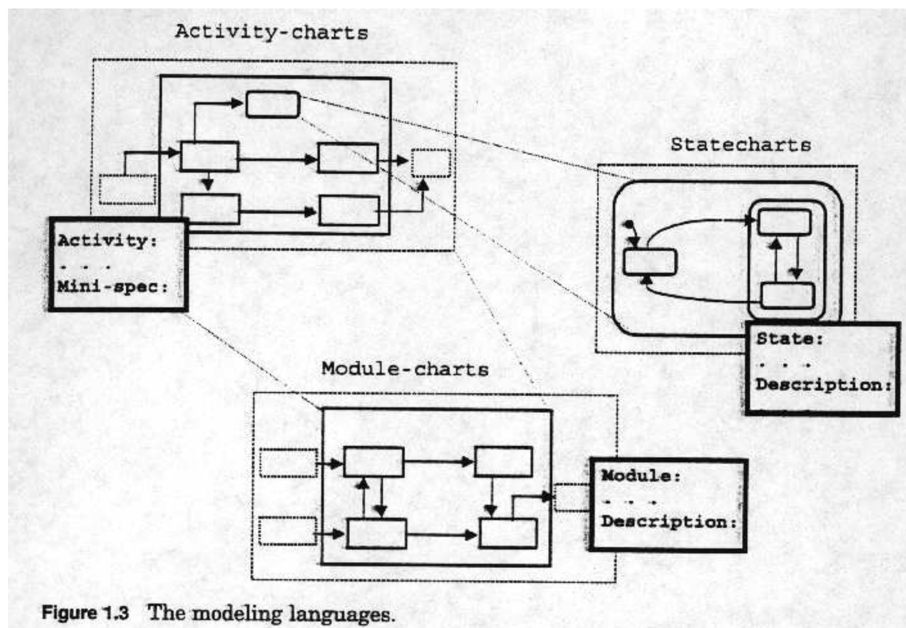


Figure 1.3 The modeling languages.

1.1.1 Functional Decomposition

- In the Statechart approach, the functionality of a system is described by **functional decomposition**, by which a system is viewed as a collection of interconnected functional components, called **activities**, organized into a hierarchy.

- E.g., in the activity-chart EWS_ACTIVITIES, the SET_UP components can be decomposed leading to a multi-level decomposition of EWS_ACTIVITIES:

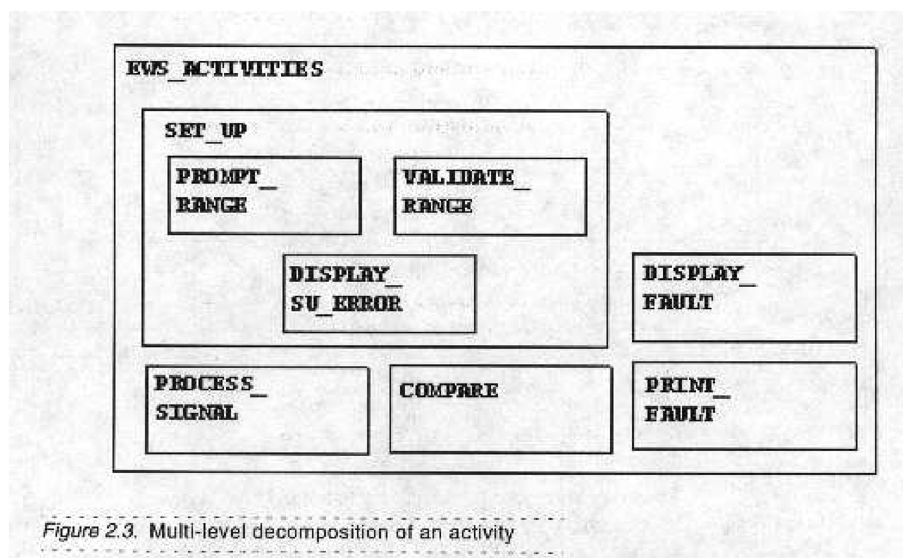


Figure 2.3. Multi-level decomposition of an activity

- Each of the activities may be decomposed into subactivities repeatedly until the system is specified in terms of **basic activities**.

  There are specified using textual description (formal or

informal), or code in a programming language, inside the Data Dictionary.

- The intended meaning of the functional decomposition is that **the capabilities of the parent activity are distributed between its subactivities**.

- The **order** in which these subactivities are performed, and the **conditions** that cause their **activation** or **deactivation** are not represented in the functional view and are specified in the **behavioral view**, i.e., in the (one) statechart associated with the parent activity-chart.

- Please observe that a functional component may very well be **reactive** in nature (cfr. the first session lecture).

- Activities can represent **objects, processes, functions, logical machines**, or any other kind of **functionally distinct entity**.

- In the following sections we'll confine ourselves to **function-based decomposition** of an activity-chart. We shall not discuss **object-based decomposition** (see Section 2.1.3 of Harel & Politi)

1.1.2 Function-based decomposition of activity-charts

- In function-based decomposition, the activities are (possibly reactive) functions.
- As an illustration consider the EWS example.
- Its first description is in natural language:

The EWS receives a signal from an external sensor. When the sensor is connected, the EWS processes the signal and checks if the resulting value is within a specified range. If the value of the processed signal is out of range, the system issues a warning message on the operator display and posts an alarm. If the operator does not respond to this warning within a given time interval, the system prints a fault message on a printing facility and stops monitoring the signal. The range limits are set by the operator. The system becomes ready to start monitoring the signal only after the range limits are set. The limits can be re-defined after an out-of-range situation has been detected, or after the operator has deliberately stopped the monitoring.

- Next we decompose this narrative to describe its functionality:

  - **The EWS receives a signal from an external sensor.**
  - **It samples and processes the signal continuously, producing some result.**
  - **It checks whether the value of the result is within a specified range that is set by the operator.**
  - **If the value is out of range, the system issues a warning message on the operator display and posts an alarm.**
  - **If the operator does not respond within a given time interval, the system prints a fault message on a printing facility and stops monitoring the signal.**

- Thirdly, we identify the various functions that are described by there requirements:

SET_UP: receives the range limits from the operator.

PROCESS_SIGNAL: reads the "raw" signal from the sensor and performs some processing to yield a value that is to be compared to the range limits.

COMPARE: compares the value of the processed signal with the range limits.

DISPLAY_FAULT: issues a warning message on the operator display and posts an alarm.

PRINT_FAULT: prints a fault message on the printing facility.

- Notice that this description also contains info about handled data. An activity may **transform** its input into output to be consumed by other functions, which are **internal** or **external** to the system:

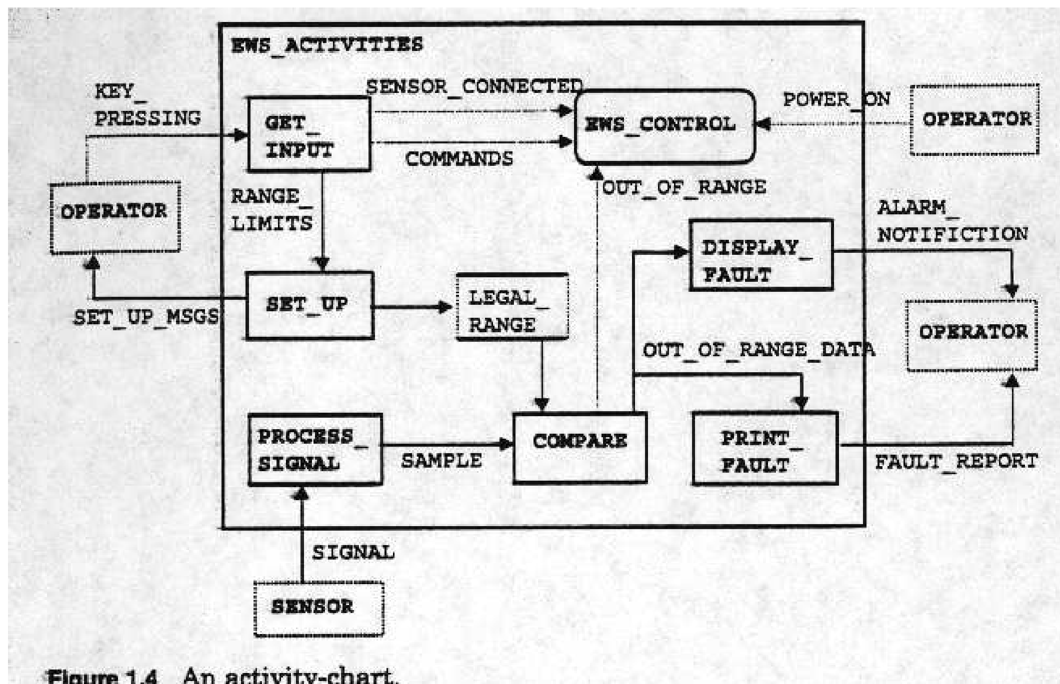


Figure 1.4 An activity-chart.

- The interface of an activity is described in terms of input and output signals, both data and control, see last figure.

1.1.3 System context

One of the first decisions to be made when developing a system involves its **boundaries**, or, **context**. I.e., one must determine which entities are part of the environment of a system, and how they communicate with the system. The latter are called **external activities** of the system.

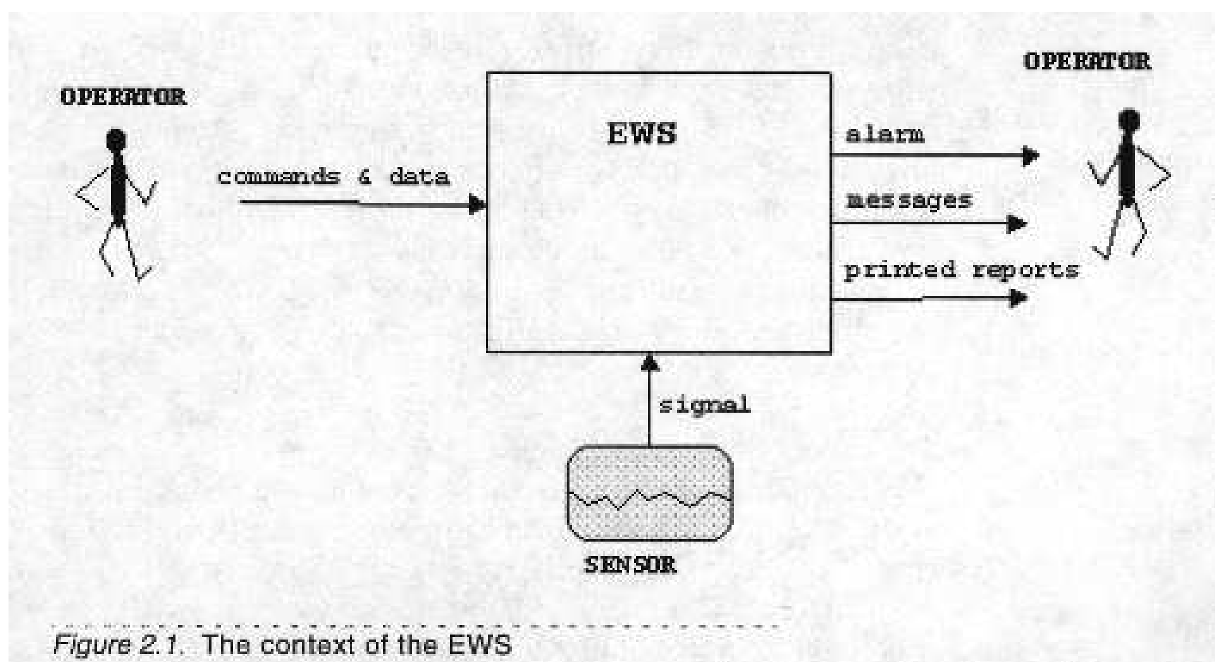


Figure 2.1. The context of the EWS

Notice that for the EWS one might have chosen for the printer to be external, leading to printer as external activity.

Different occurrences of the same entity (here: operator) denote the same entity; these are multiplicated of ease of drawing.

1.1.4 Decomposition process

The functional view is specified by **Activity-charts**, together with a **Data Dictionary** that contains additional information about the elements appearing in the charts, e.g., about their basic activities.

1.2 Activities and their representation

We continue the functional decomposition of the EWS, started with:

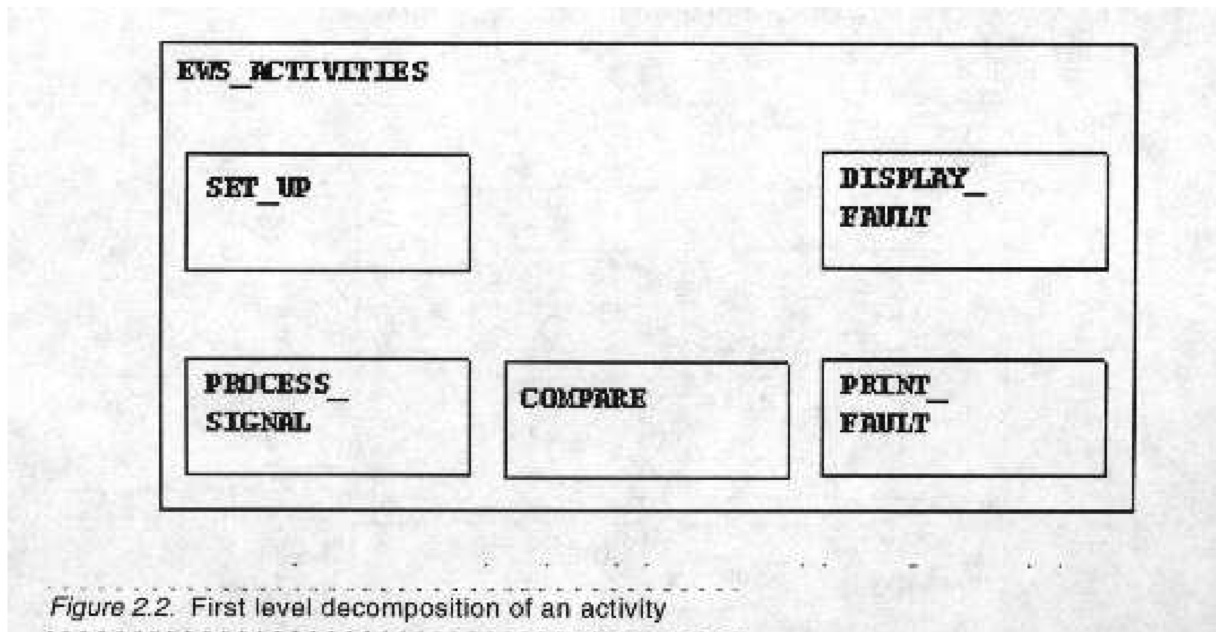


Figure 2.2. First level decomposition of an activity

This activity chart contains one top-level box, representing the **top-level activity** of the chart.

On their turn, the activities appearing above can be decomposed themselves, as SET_UP:

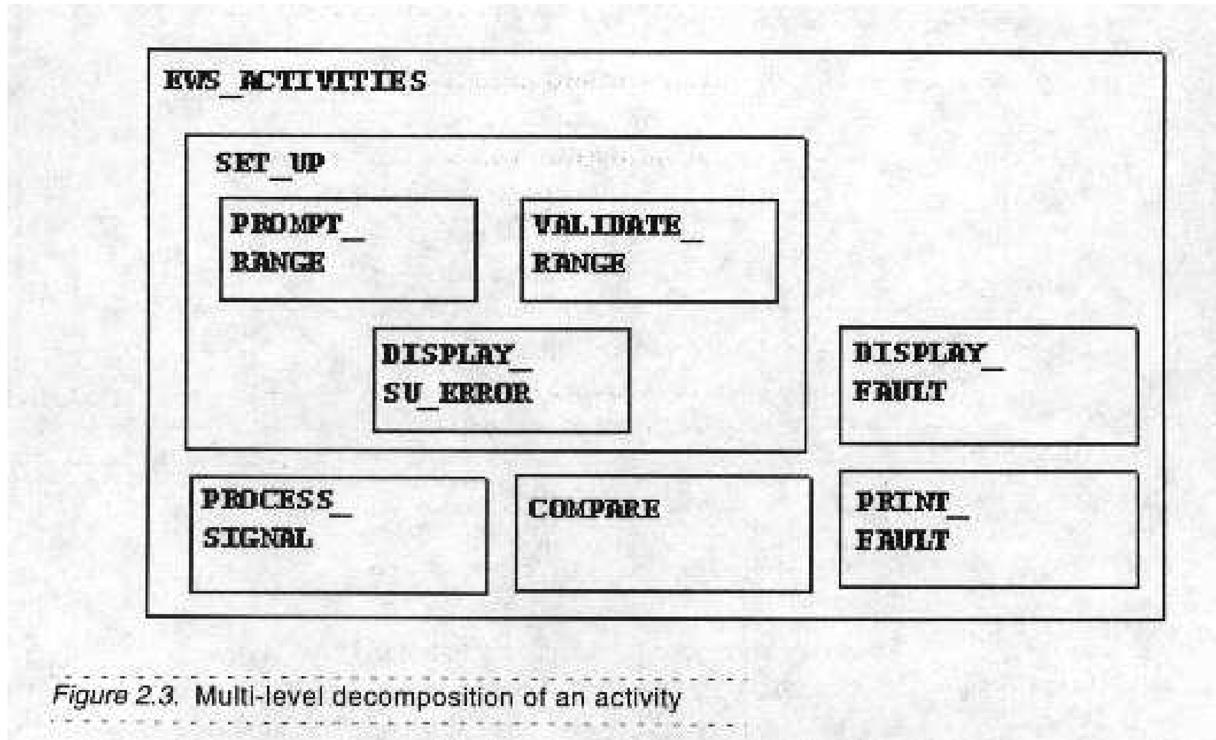


Figure 2.3. Multi-level decomposition of an activity

Some terminology

- EWS_ACTIVITIES is called **top-level activity**
- EWS_ACTIVITIES is also called **parent activity** of SET_UP, COMPARE, etc., which are called **descendants** of EWS_ACTIVITIES, as are the subactivities PROMPT_RANGE etc. of SET_UP, who have SET_UP and EWS_ACTIVITIES as **ancestor**.

Each activity has a corresponding item in the Data Dictionary, which may contain additional information.

1.3 Flow of Information between Activities

- Consider the following chart:

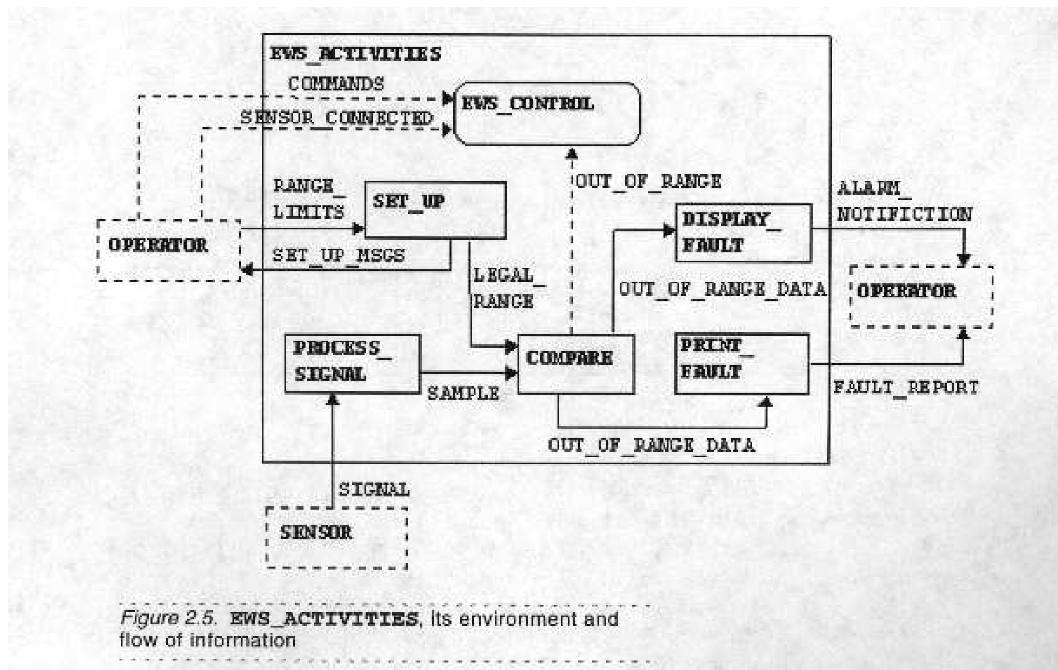


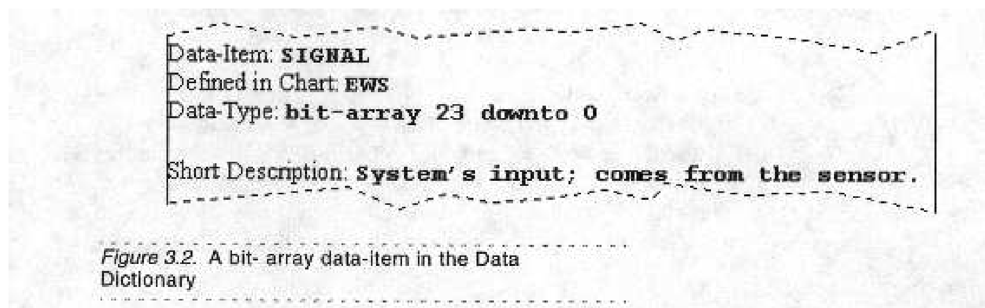
Figure 2.5. **EWS_ACTIVITIES**, Its environment and flow of information

- OPERATOR and SENSOR are external activities, drawn using dotted lines.
- Different occurrences of OPERATOR refer to the same entity.
- Solid arrows denote **data-flow-lines** between activities.
- Control of EWS_ACTIVITIES is handled in its **control activity chart** EWS_CONTROL, a statechart (drawn using rounded corners).
- Dotted arrows denote **control-flow-lines**, carrying info or signals used in making **control decisions**.

1.3.1 Flow lines

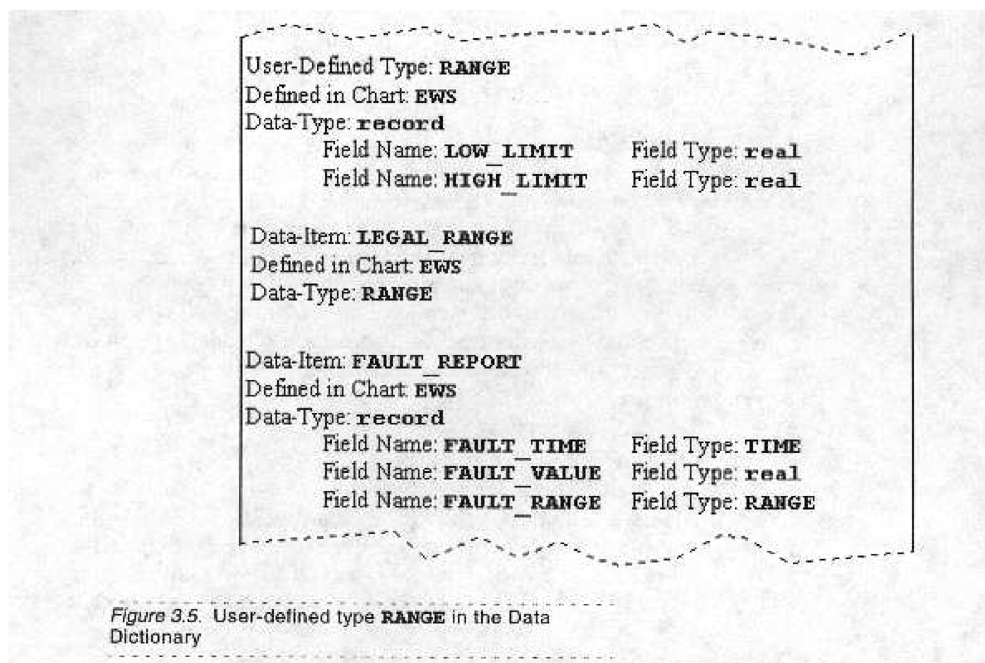
A label on a flow line denotes:

- Either a single information element that flows along the line, i.e., a **data-item**, **condition**, or **event**.
- Or a group of such elements, as in, e.g.:


Data-Item: SIGNAL
Defined in Chart: EWS
Data-Type: bit-array 23 downto 0

Short Description: System's input; comes from the sensor.

Figure 3.2. A bit- array data-item in the Data Dictionary


User-Defined Type: RANGE
Defined in Chart: EWS
Data-Type: record
Field Name: LOW_LIMIT Field Type: real
Field Name: HIGH_LIMIT Field Type: real

Data-Item: LEGAL_RANGE
Defined in Chart: EWS
Data-Type: RANGE

Data-Item: FAULT_REPORT
Defined in Chart: EWS
Data-Type: record
Field Name: FAULT_TIME Field Type: TIME
Field Name: FAULT_VALUE Field Type: real
Field Name: FAULT_RANGE Field Type: RANGE

Figure 3.5. User-defined type RANGE in the Data Dictionary

Such a group is called **information-flow**.

- A flow-line originates from its **source activity**, and leads to its **target activity**:

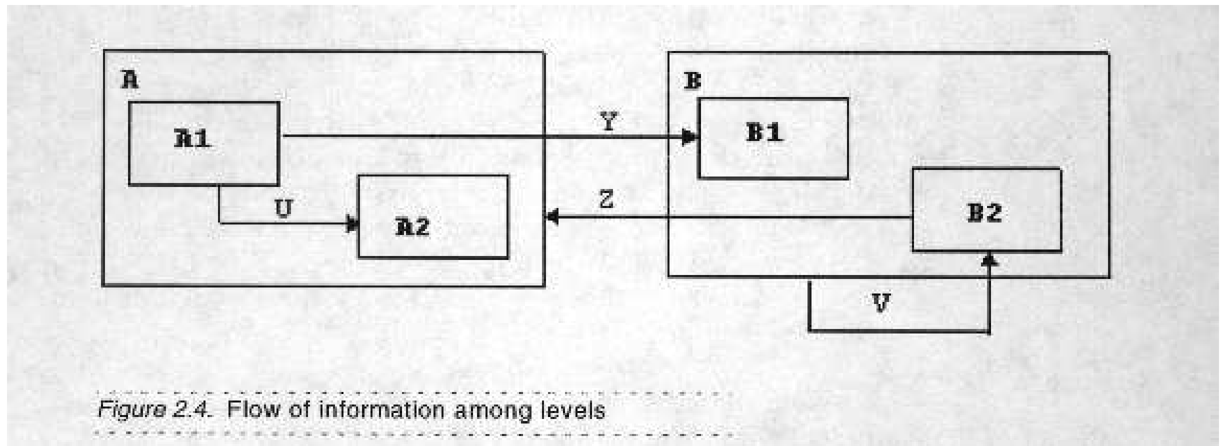


Figure 2.4. Flow of information among levels

- An arrow can be connected to a non-basic box, meaning it relates to **all the subboxes** within the box in question, see above the data flow lines labeled V and Z.

- Information flow SIGNAL in Figure 2.5 is declared in the Data Dictionary as in Figure 3.2 and is used in data processing.

- Information flow COMMANDS in the Data Dictionary declared as below, is used to denote **control issues**.

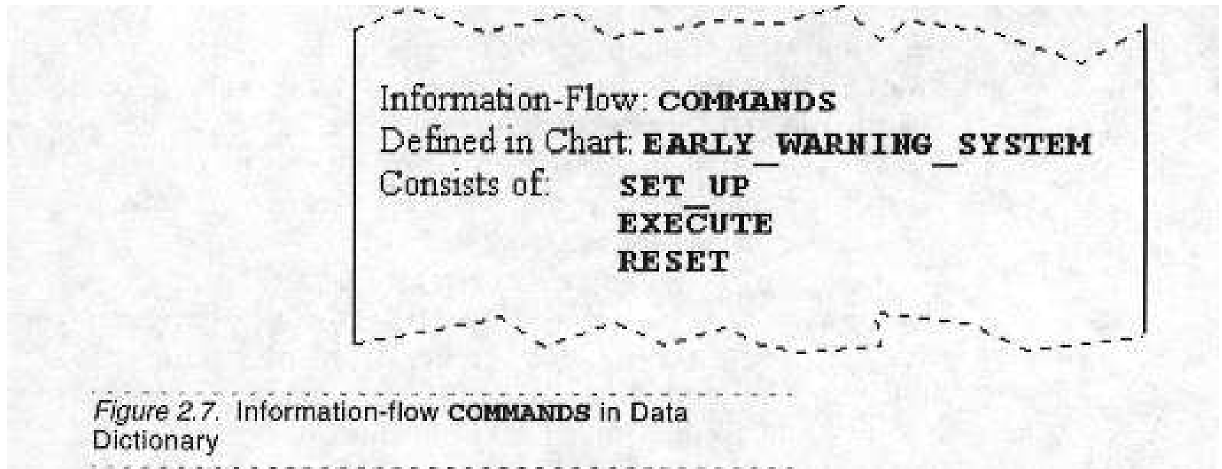


Figure 2.7. Information-flow **COMMANDS** in Data Dictionary

- Flow lines may represent, e.g.,
  - parameter passing to procedures
  - passing of values of global variables
  - messages transferred in distributed systems
  - queues between tasks in real-time applications
  - signals flowing along physical links in hardware systems

- Flows can be continuous or discrete in time.

1.3.2 Flowing elements

- Three types of information elements flow between activities: **events, conditions, data-items**.
- Their differences are in their **domain of values** and **timing characteristics**:

  Events are instantaneous signals used for synchronization purposes, e.g., OUT_OF_RANGE in Figure 2.5.

  Conditions are **persistent** signals that are either true or false, e.g., SENSOR_CONNECTED in Figure 2.5.

  Data-item are **persistent** and may hold values of various types and structures, e.g., SIGNAL, a **bit-array**, or LEGAL_RANGE, a **record** with two fields of type **real**, HIGH_LIMIT and LOW_LIMIT.
- All three types of information elements can be arranged in array and record structures:

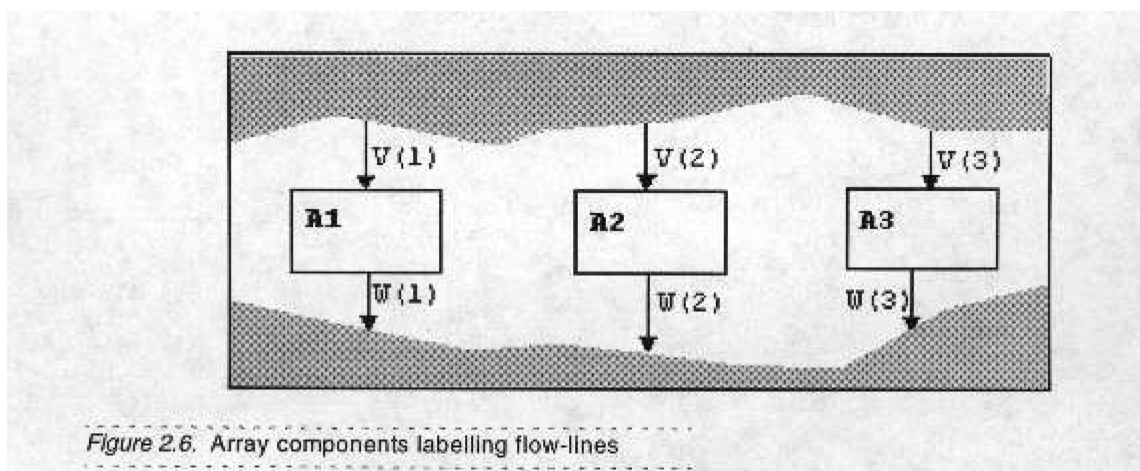


Figure 2.6. Array components labelling flow-lines

1.3.3 Information Flows

The number of lines in an activity chart can be reduced by grouping information elements into an **information-flow**, used to label a common flow line, see e.g. COMMANDS in the following figure, consists of SET_UP, EXECUTE, RESET.

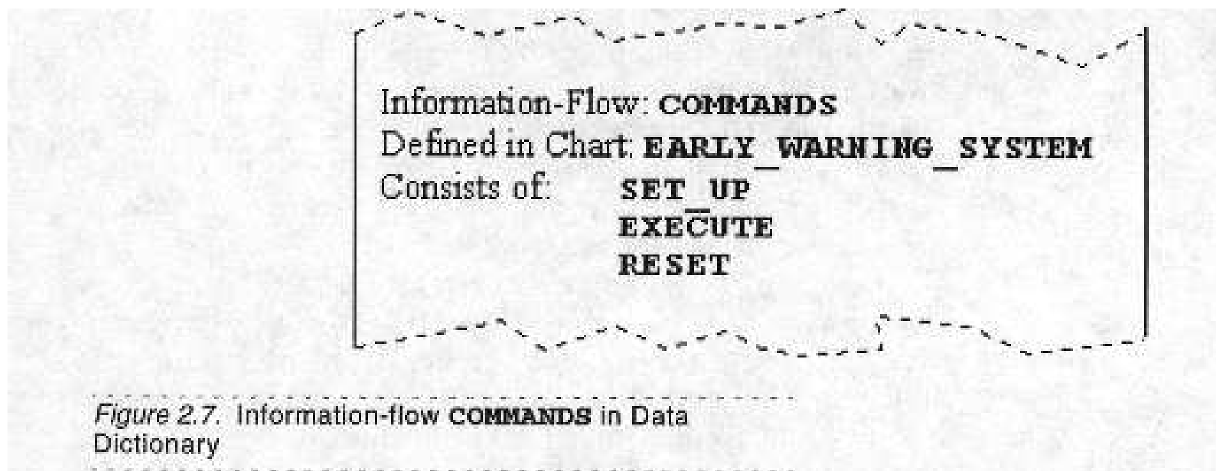


Figure 2.7. Information-flow **COMMANDS** in Data Dictionary

1.3.4 Data Stores

- There are no restrictions on the time that data reside on a flow line. Nevertheless it is often more natural to incorporate an explicit data store in the chart:

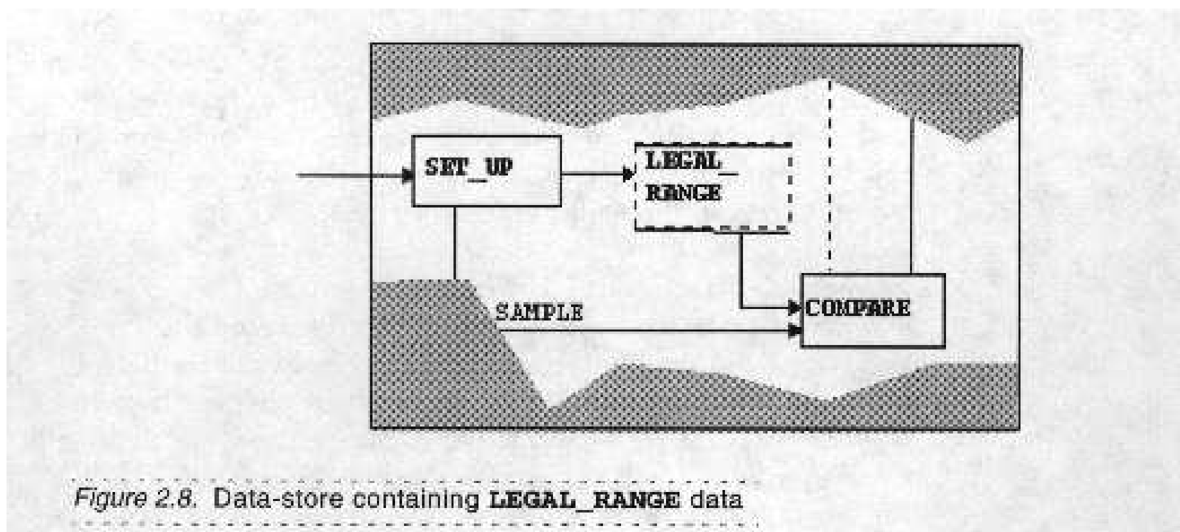


Figure 2.8. Data-store containing **LEGAL_RANGE** data

- A data item is defined in the Data Dictionary with the same name as the data store. Any structure given to a data item is inherited by the data store.

1.4 The Behavioral Functionality of Activities

- The behavior of subactivities of an activity chart is described by its **control activity**, whose function is to control their **sibling** activities (i.e., the other subactivities in the chart).

- A control activity may explicitly start and stop its sibling activities, i.e., EWS_CONTROL controls SET_UP, PROCESS_SIGNAL, and COMPARE:

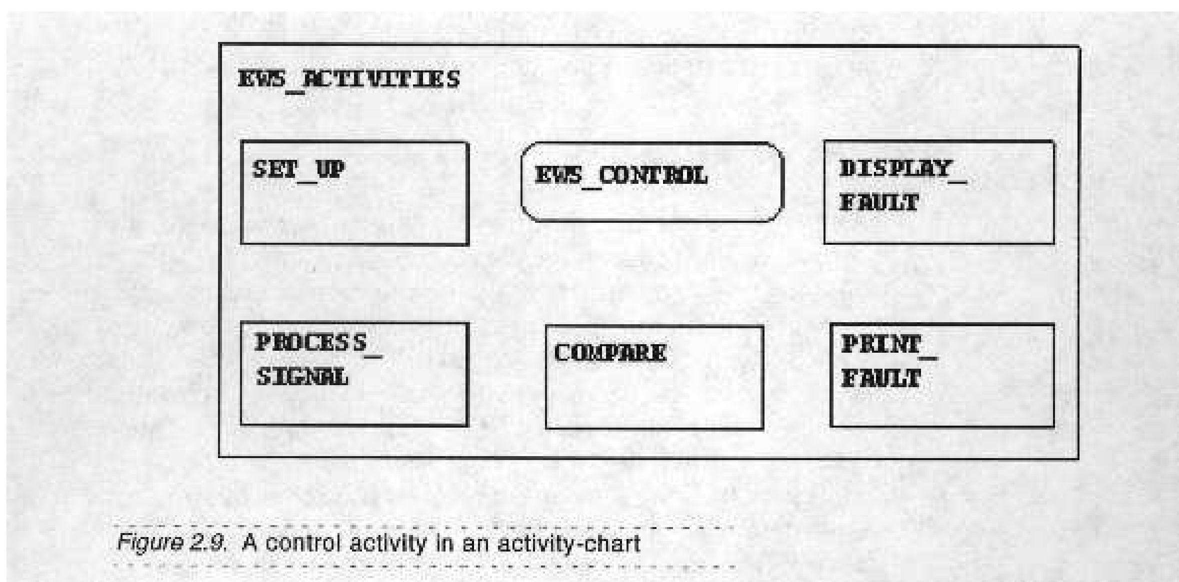
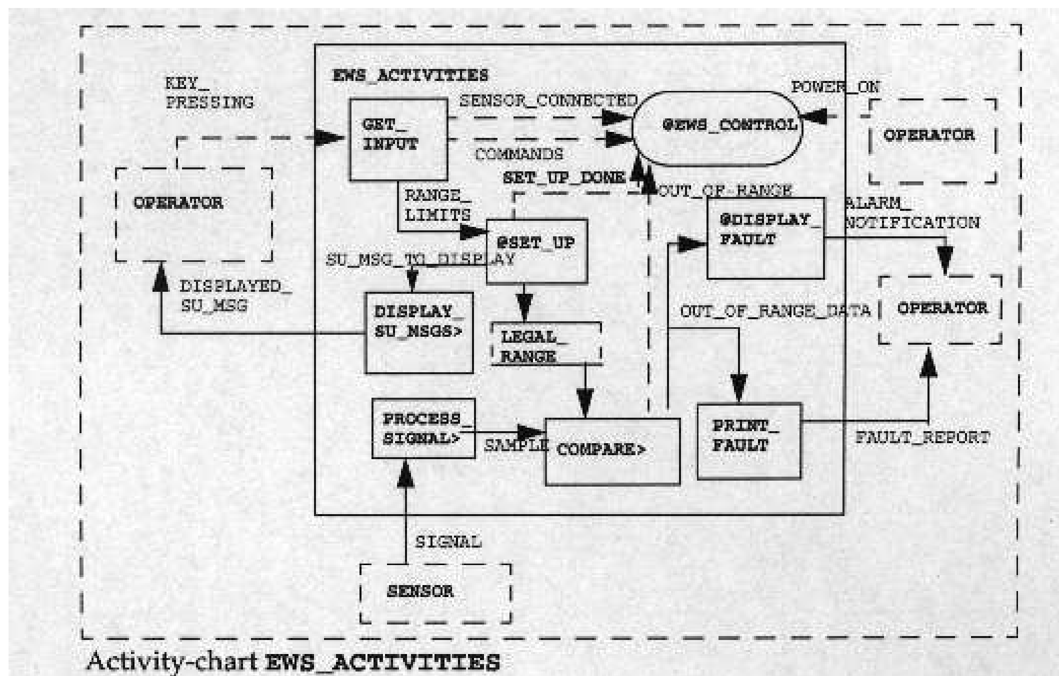


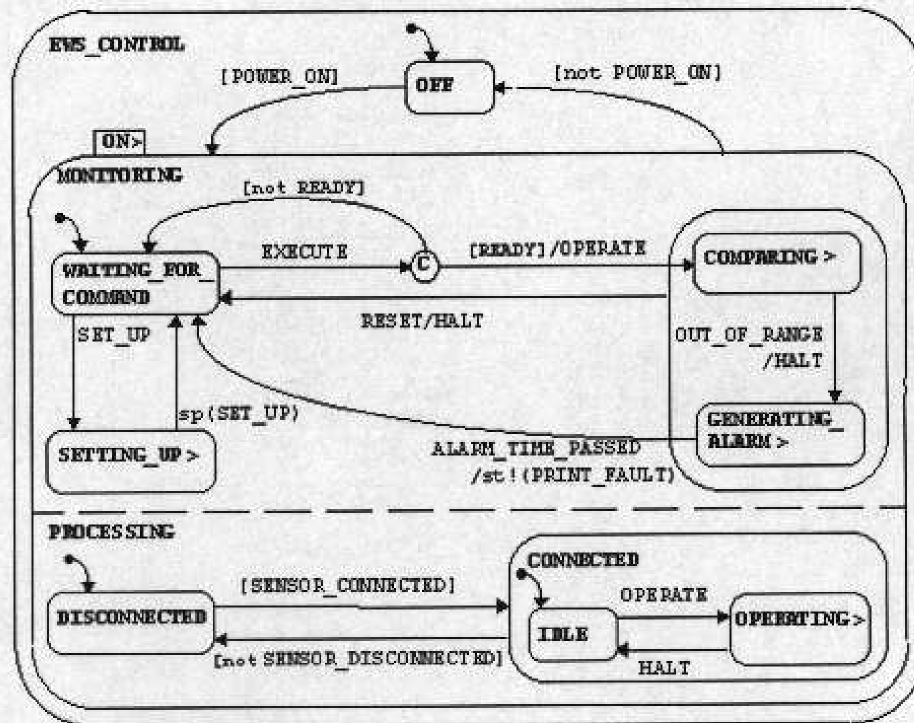
Figure 2.9. A control activity in an activity-chart

- Each activity may have **at most one** control activity.

- The control activity, depicted as a rectangle with rounded corners, cannot have subactivities. Rather its specification is that of a Statechart, see next slide.



Activity-chart **EWS_ACTIVITIES**



Statechart **EWS_CONTROL**

1.4.1 Activities in the Data Dictionary

- Every activity can be described more extensively in the Data Dictionary using **textual information**.
- **Basic activities** are described in the Data Dictionary by **executable textual descriptions**, specifying patterns of behavior. These patterns are:

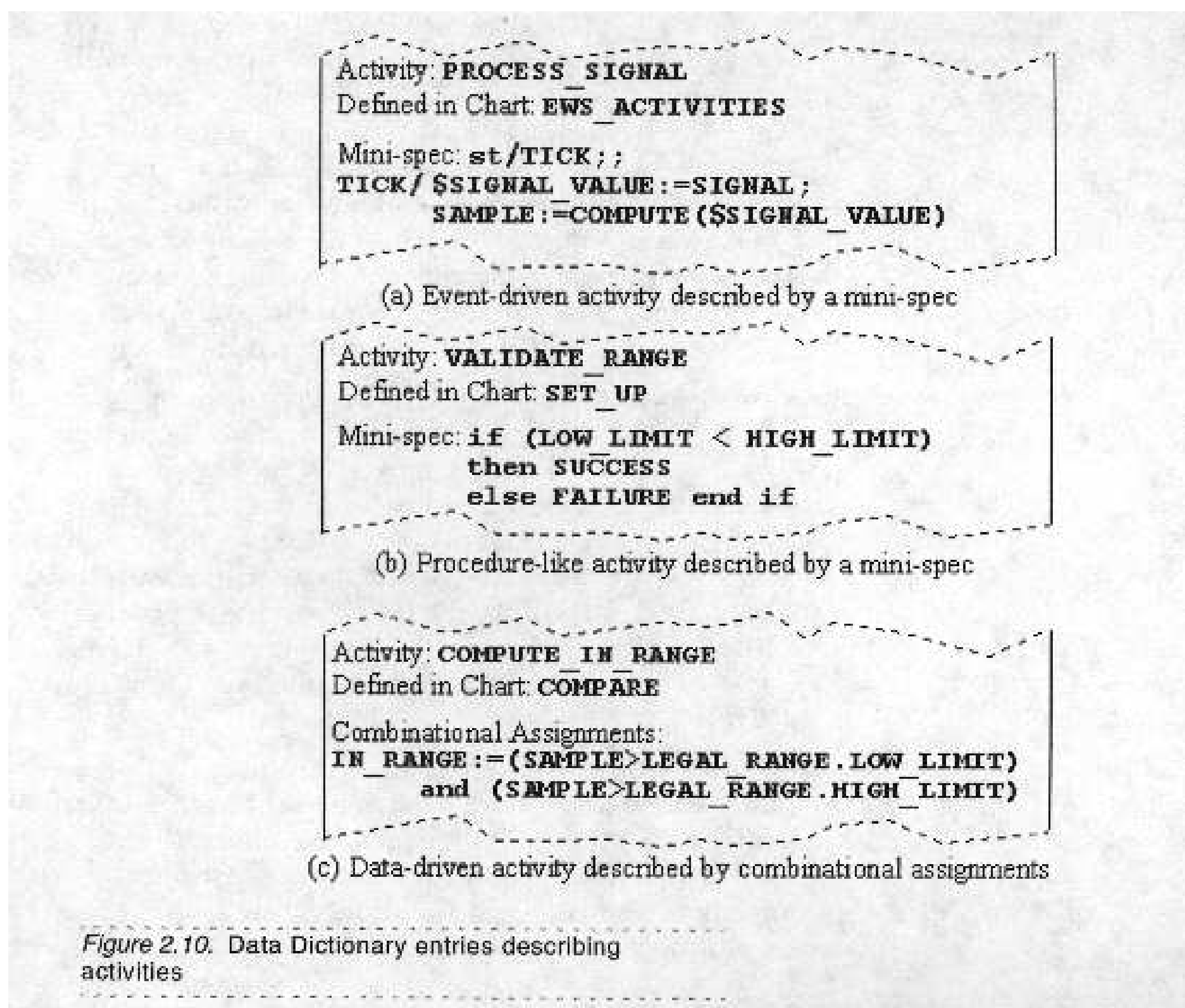


Figure 2.10. Data Dictionary entries describing activities

1.5 Connectors and Compound Flow-Lines

- The data flow lines leaving activity COMPARE in Figure 2.5 can be drawn with a **joint connector** as below:

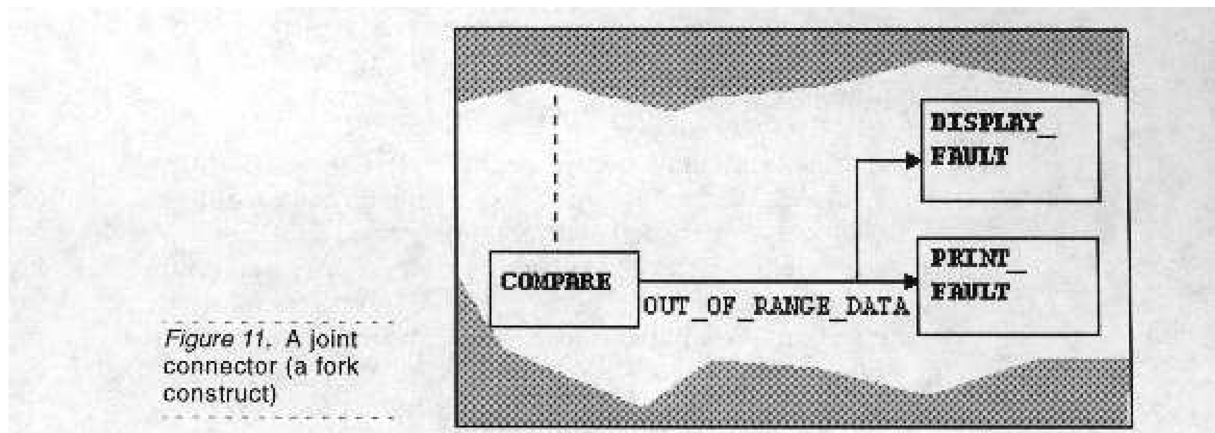


Figure 11. A joint connector (a fork construct)

Junction connectors

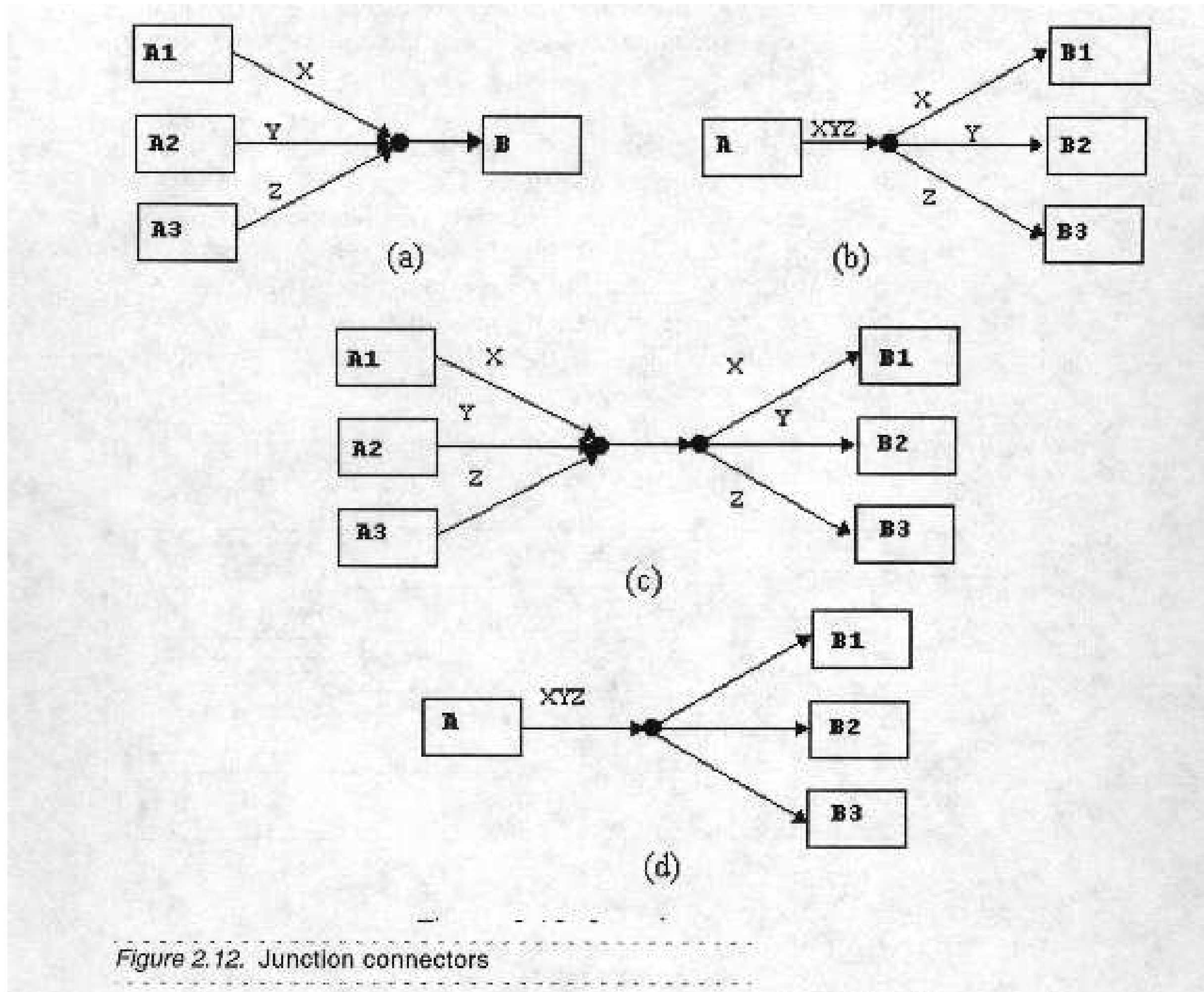


Figure 2.12. Junction connectors

Diagram connectors

- **Diagram connectors** are used when the source of a flow line is far from its target:

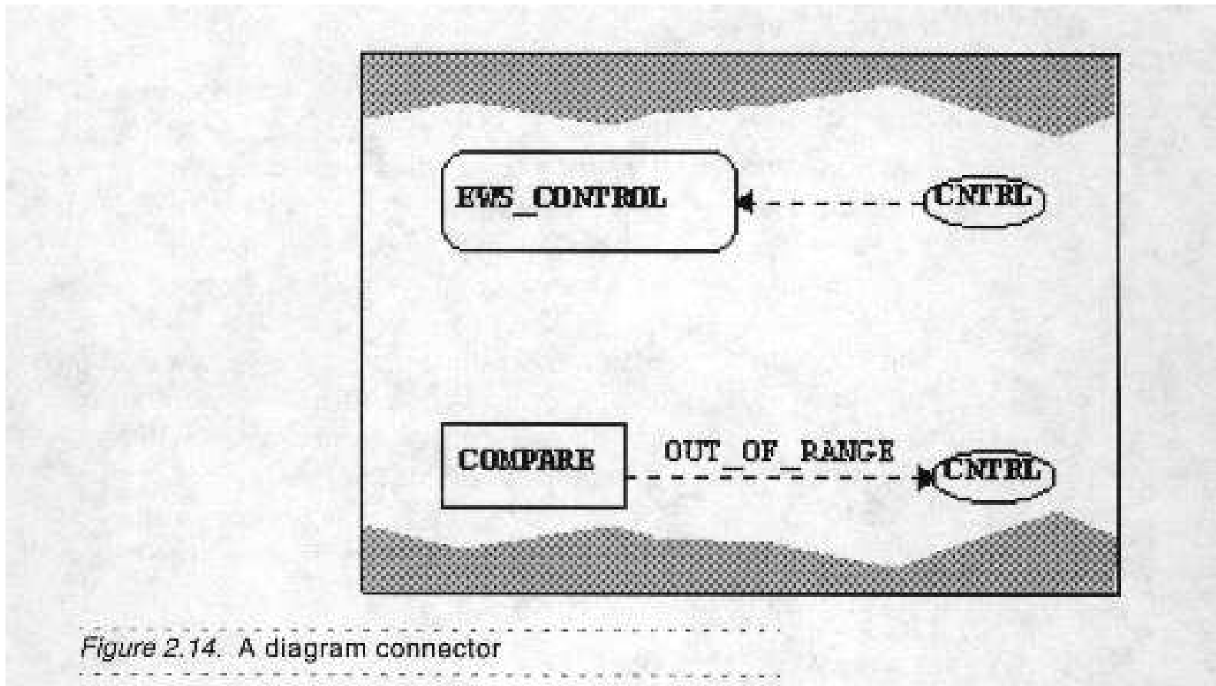


Figure 2.14. A diagram connector

Session IV

Connections between Activity-Charts and Statecharts

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

Literature: Chapters 7 and 8 of "Modeling Reactive Systems with Statecharts", by David Harel and Michal Politi. McGraw-Hill, 1998.

1.6 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.

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

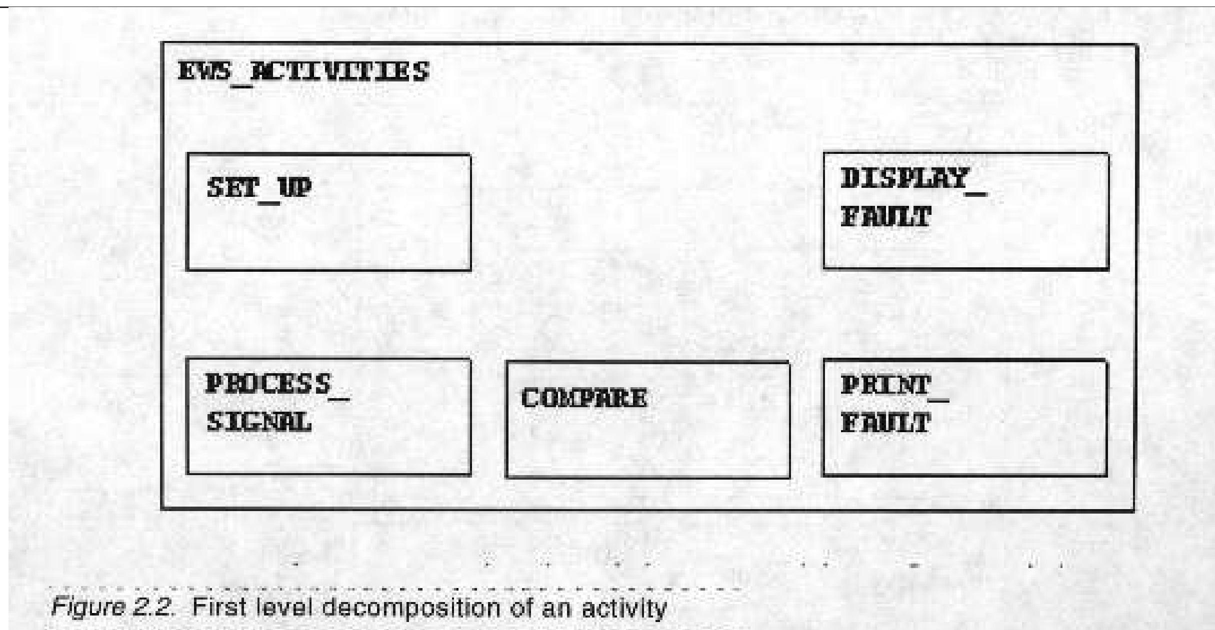


Figure 2.2. First level decomposition of an activity

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.

- One can start by carrying out a functional decomposition in activity-charts, and then add the timing and other dynamic information i 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.

1.7 Dynamics of Activities

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

1.7.1 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.

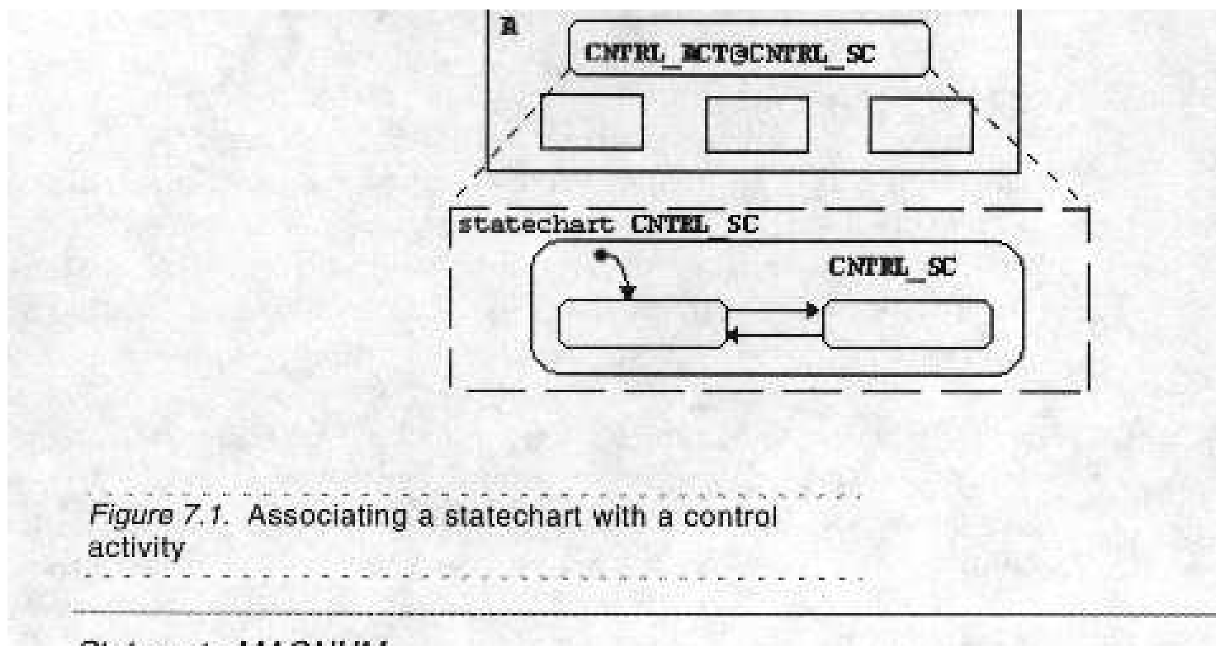


Figure 7.1. Associating a statechart with a control activity

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:

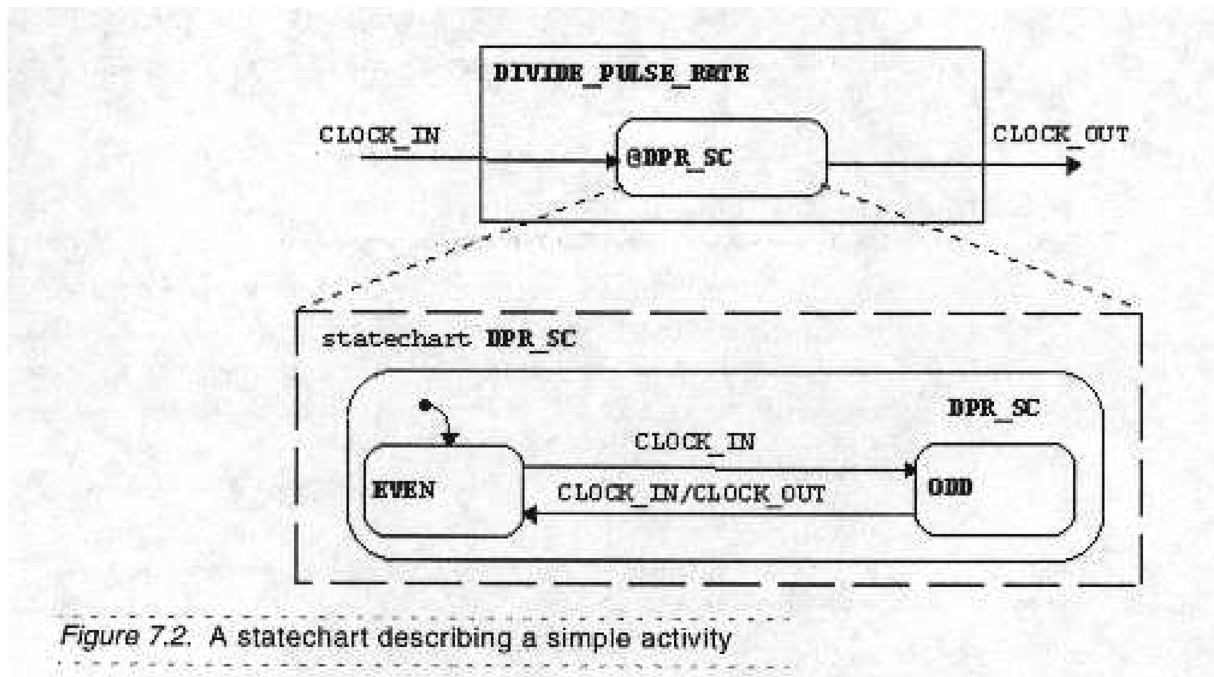


Figure 7.2. A statechart describing a simple activity

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.

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

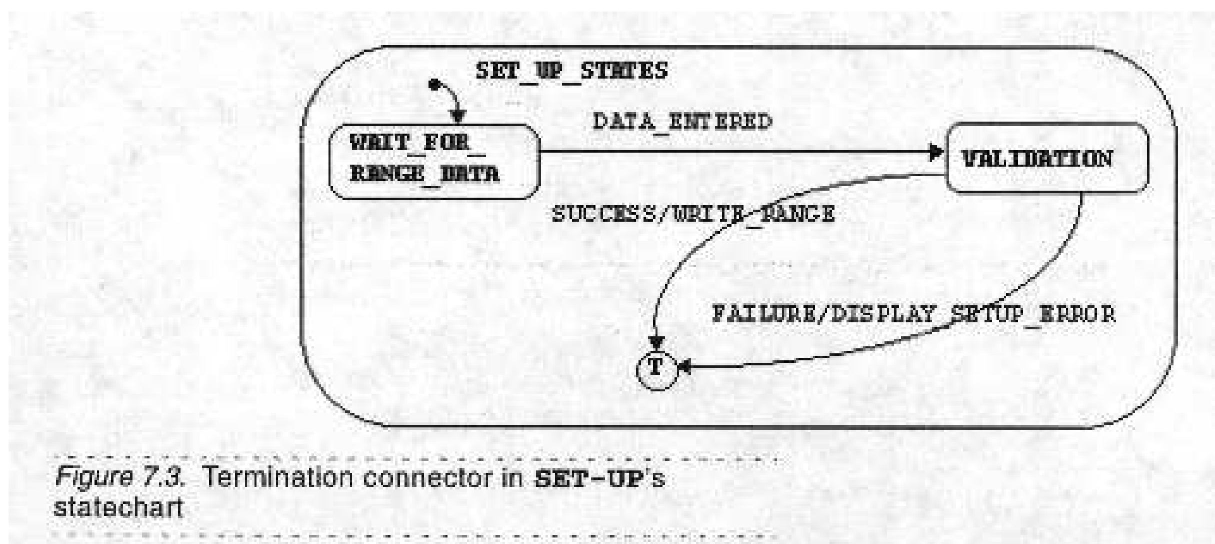


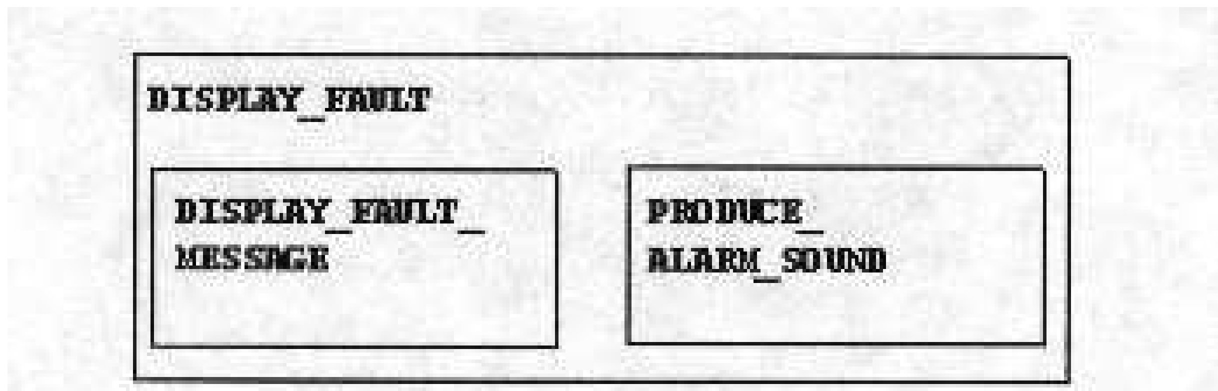
Figure 7.3. Termination connector in **SET-UP**'s statechart

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

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

1.7.3 Perpetual Activities

Sometimes there is no control activity needed:

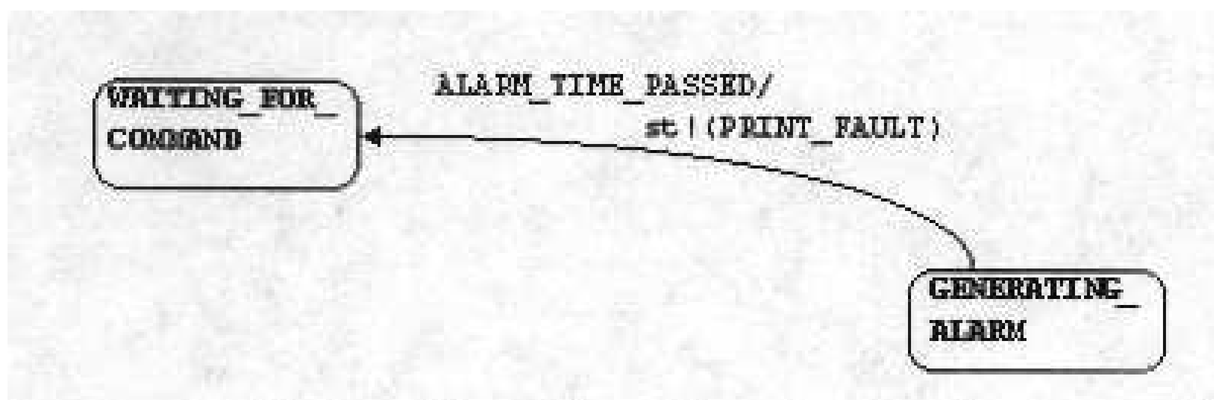


All the subactivities start when the parent activity starts, and they all stop when it stops.

1.8 Controlling the activities

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

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



Examples

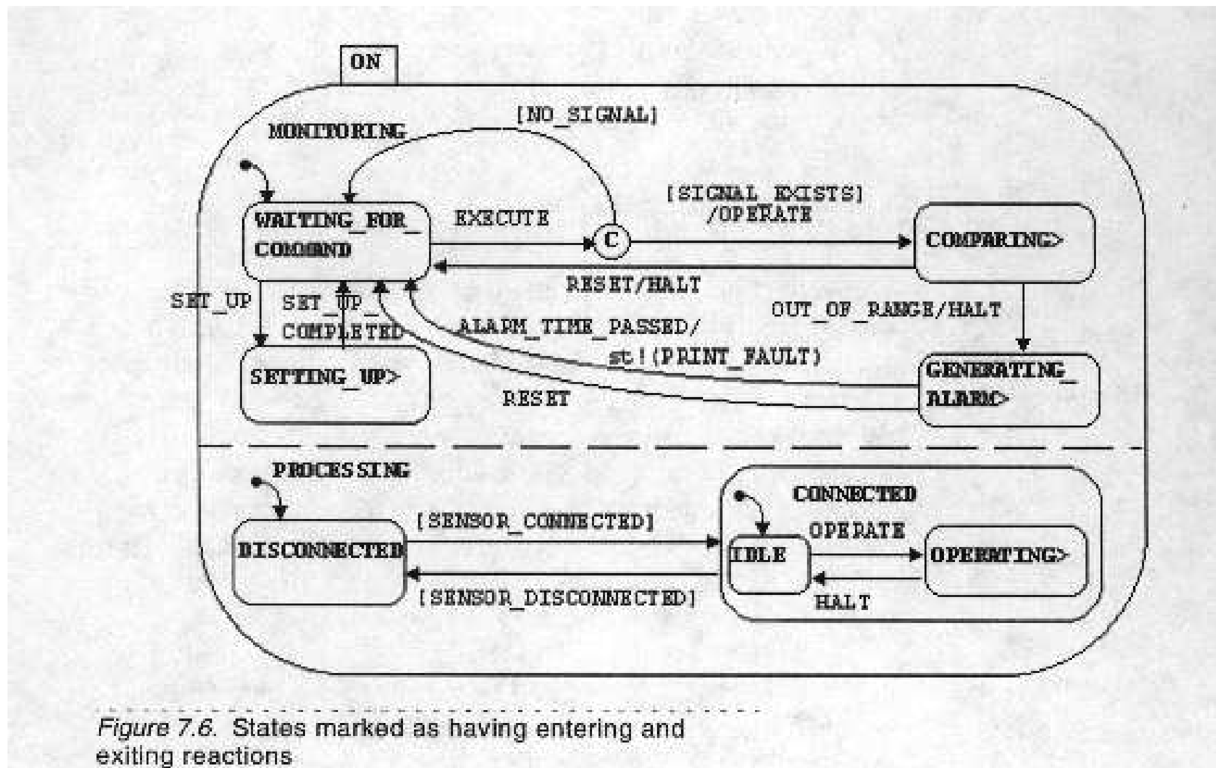


Figure 7.6. States marked as having entering and exiting reactions

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

1.8.2 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).

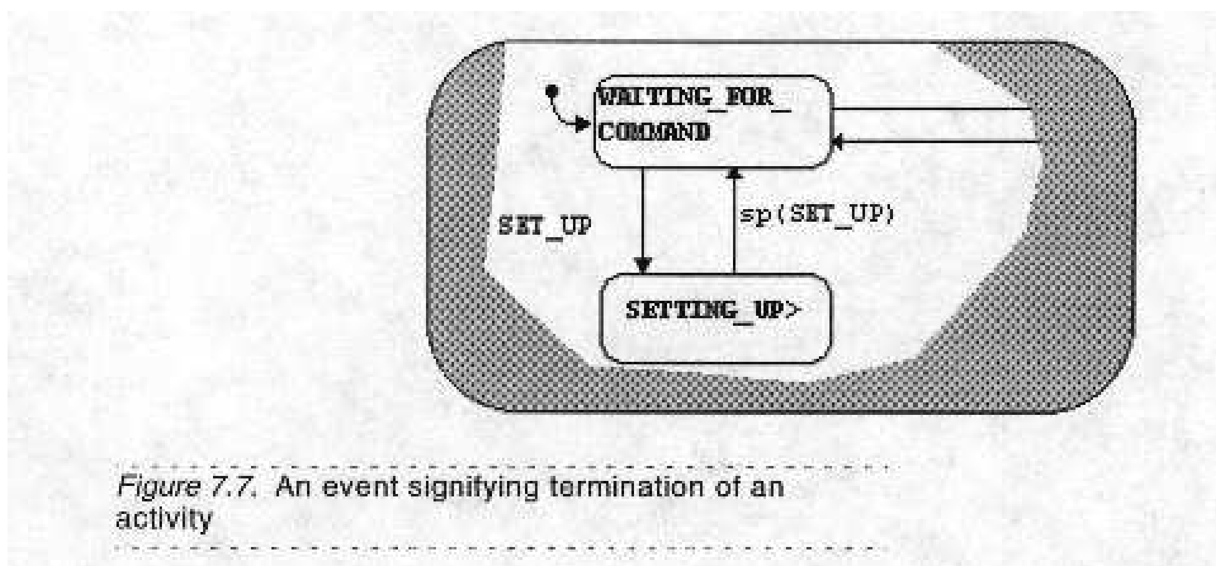


Figure 7.7. An event signifying termination of an activity

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

1.8.3 Activities Throughout and Within 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.

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

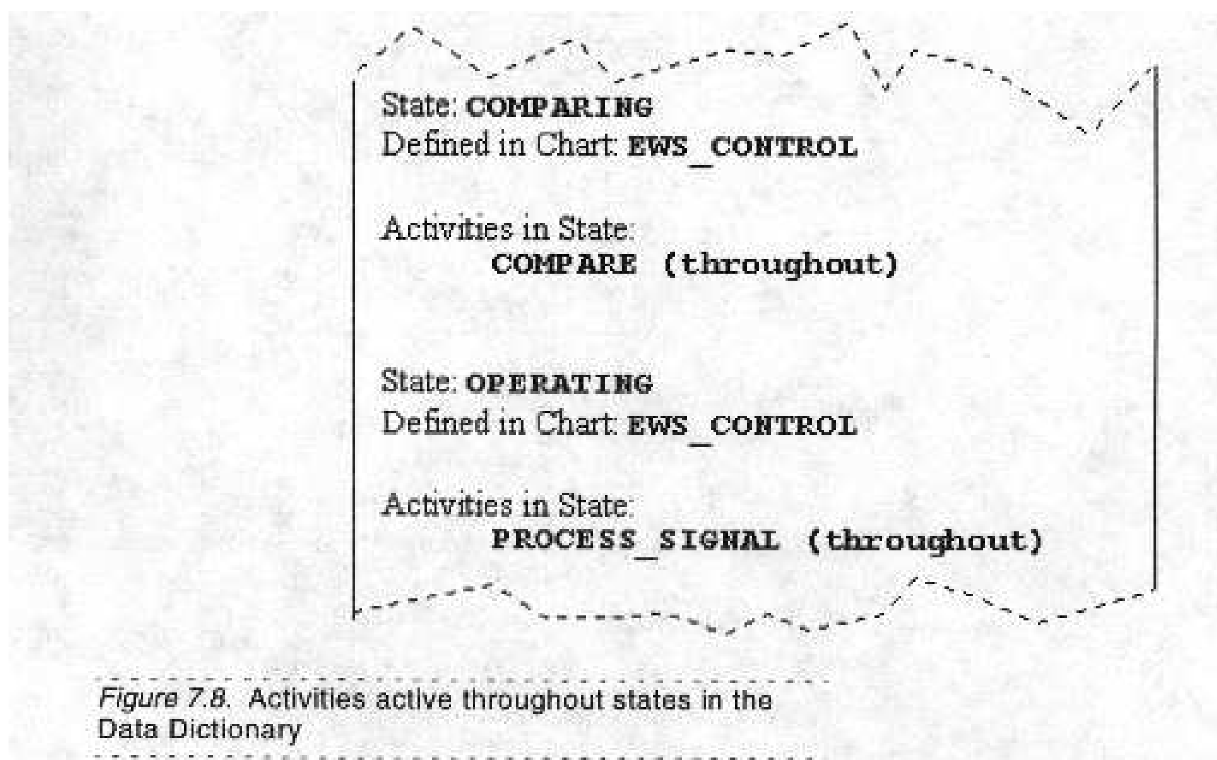


Figure 7.8. Activities active throughout states in the Data Dictionary

Another similar association is **active within**, which represents a looser connection between an activity and a state.

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

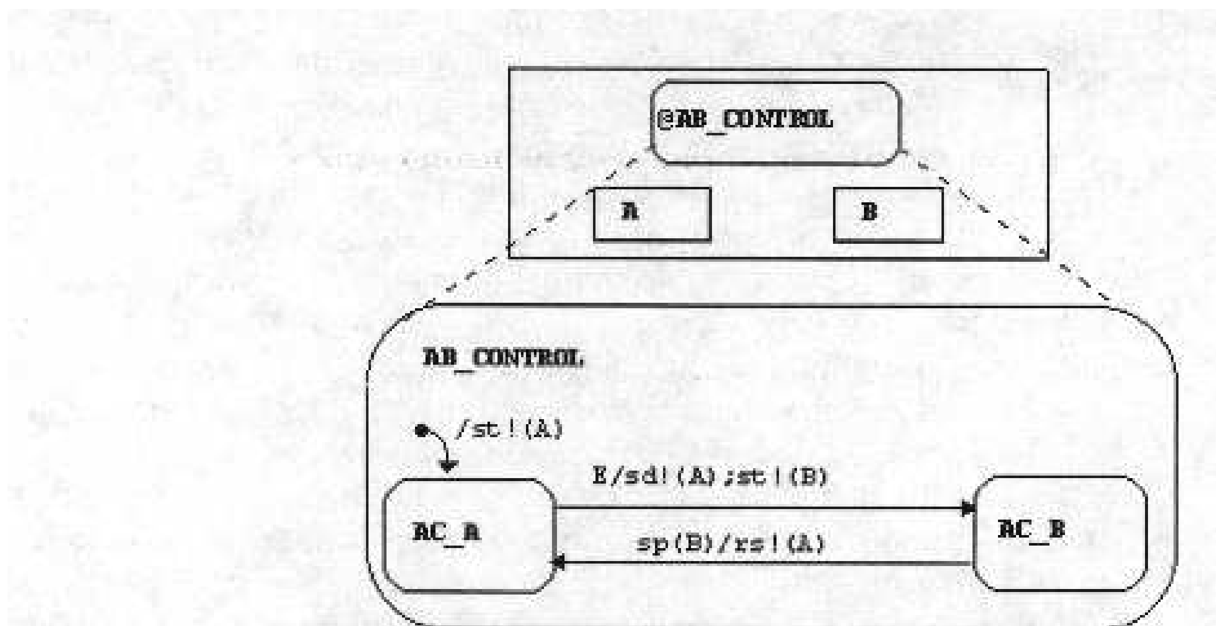


Figure 7.10. Suspending and resuming activities

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

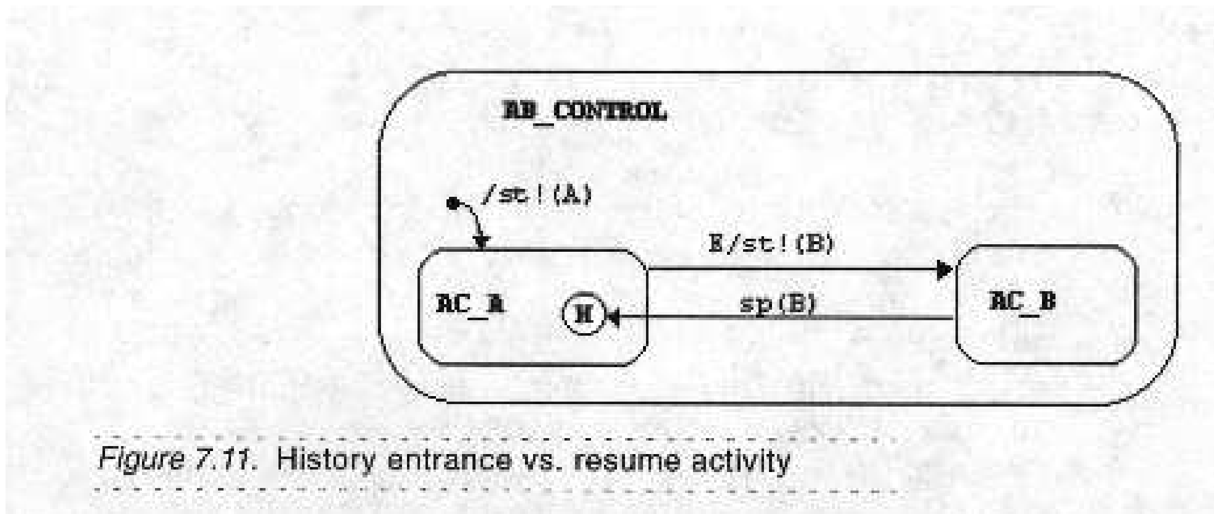


Figure 7.11. History entrance vs. resume activity

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.

1.9 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 “>”.

1.9.1 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 (;;).

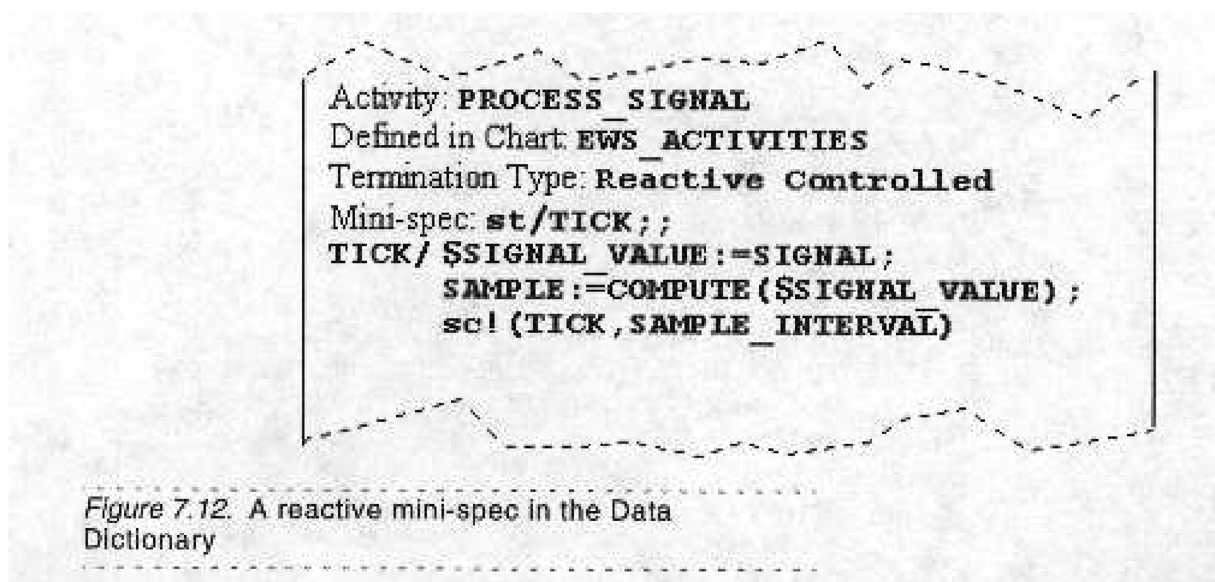
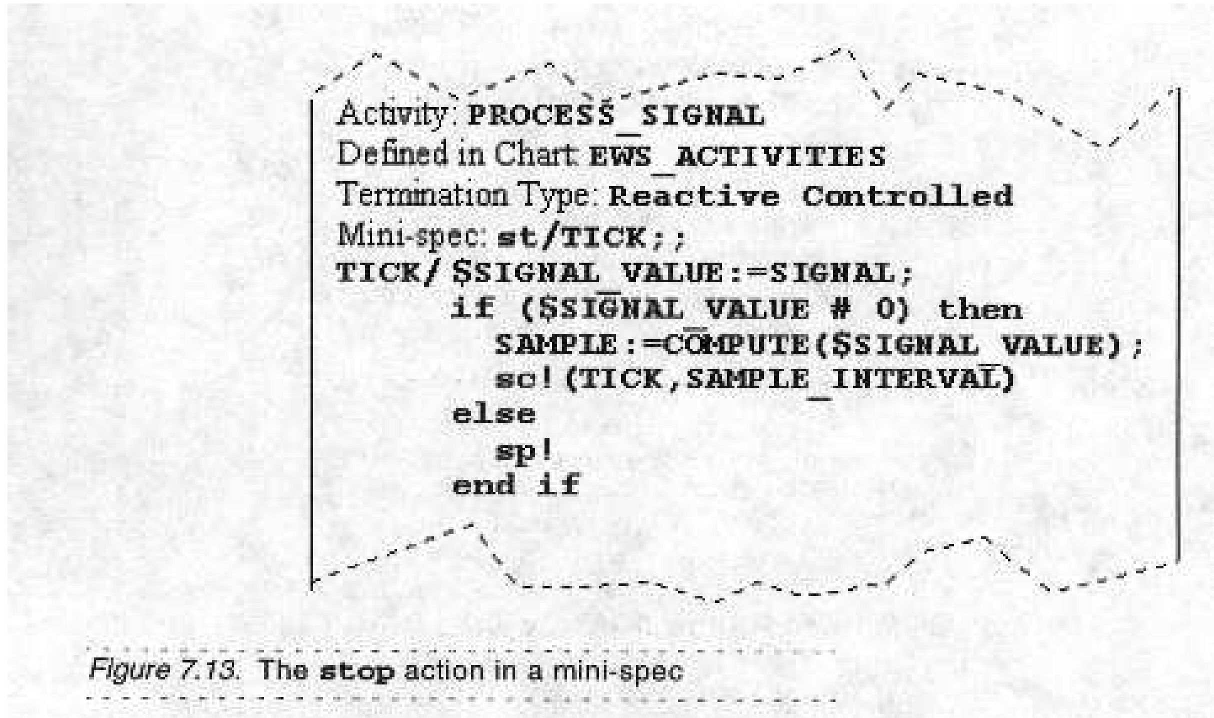


Figure 7.12. A reactive mini-spec in the Data Dictionary

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.

1.9.2 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.

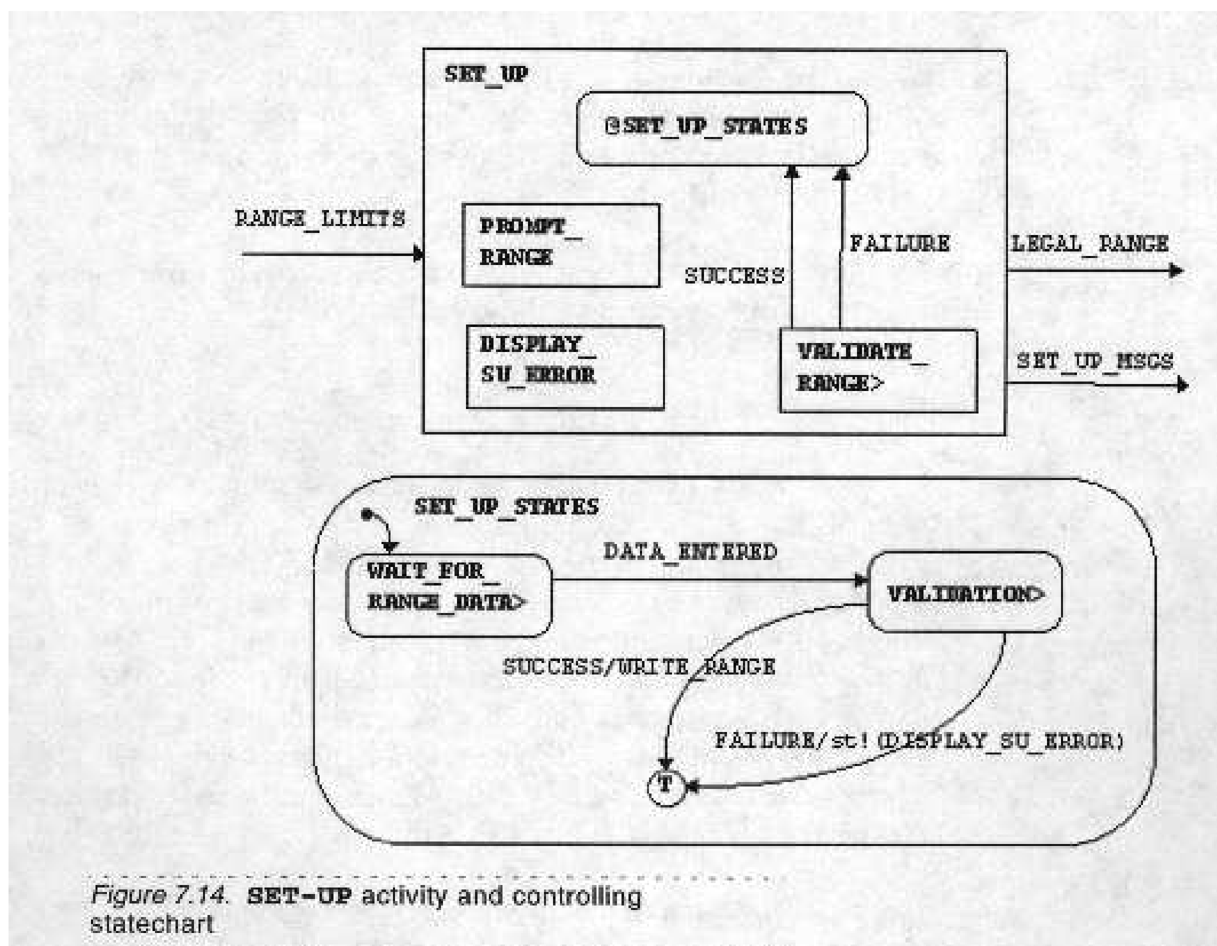


Figure 7.14. **SET-UP** activity and controlling statechart

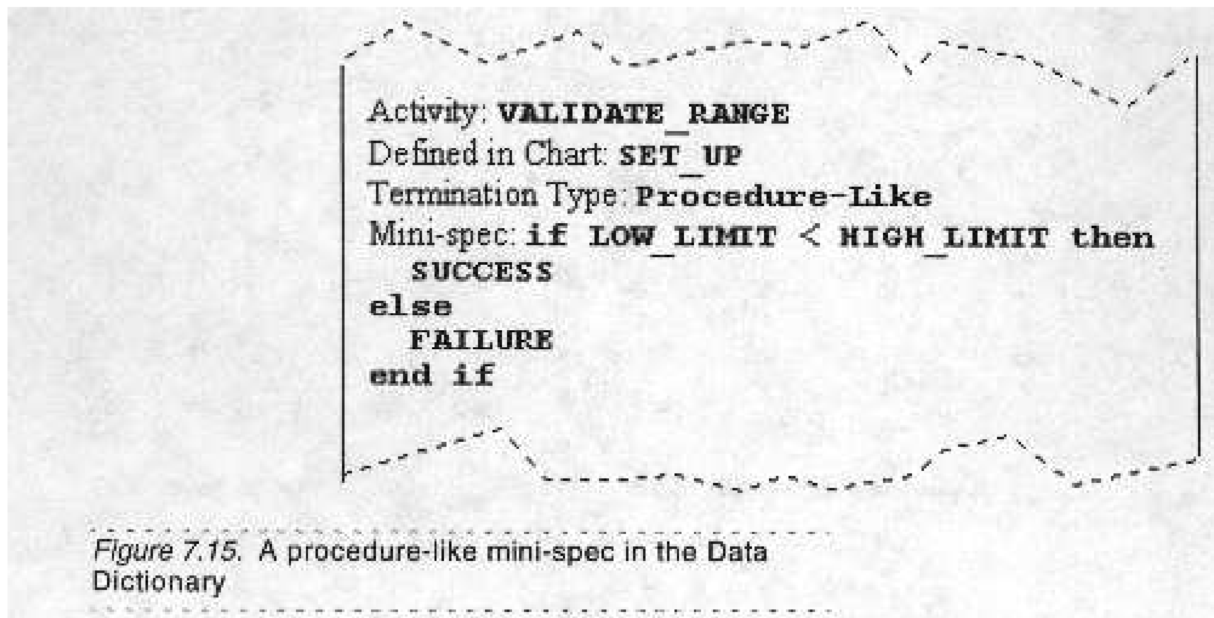


Figure 7.15. A procedure-like mini-spec in the Data Dictionary

1.9.3 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.

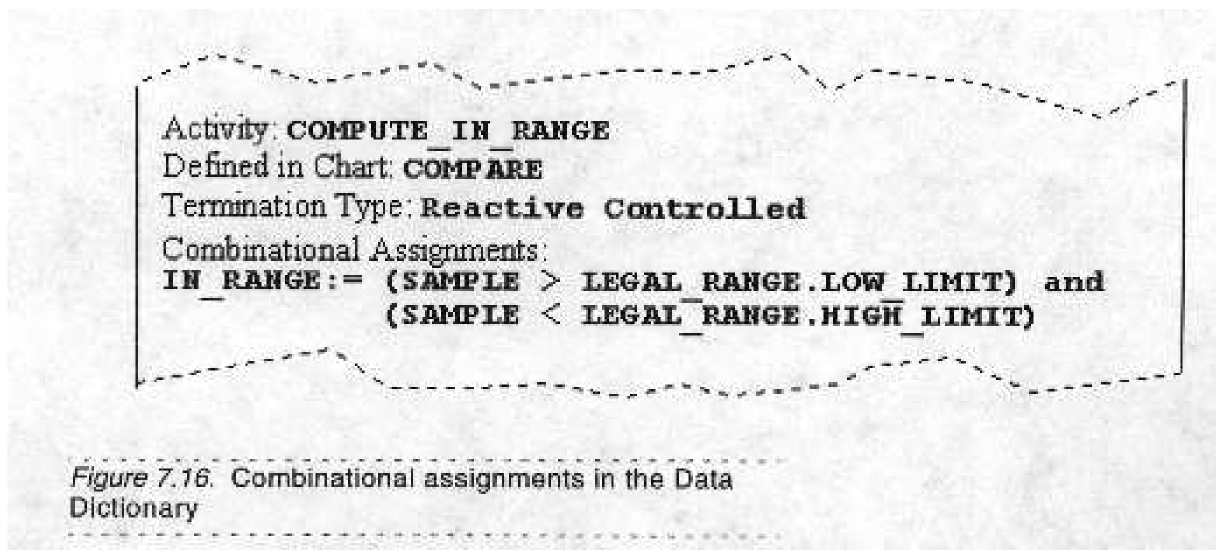


Figure 7.16. Combinational assignments in the Data Dictionary

3 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.

3.1 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.

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**

3.2 Controlling the Flow of Information

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

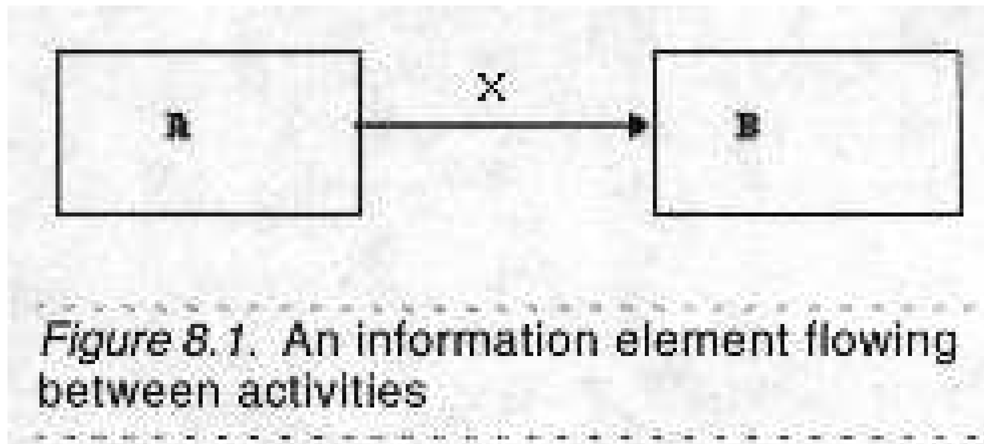


Figure 8.1. An information element flowing between activities

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

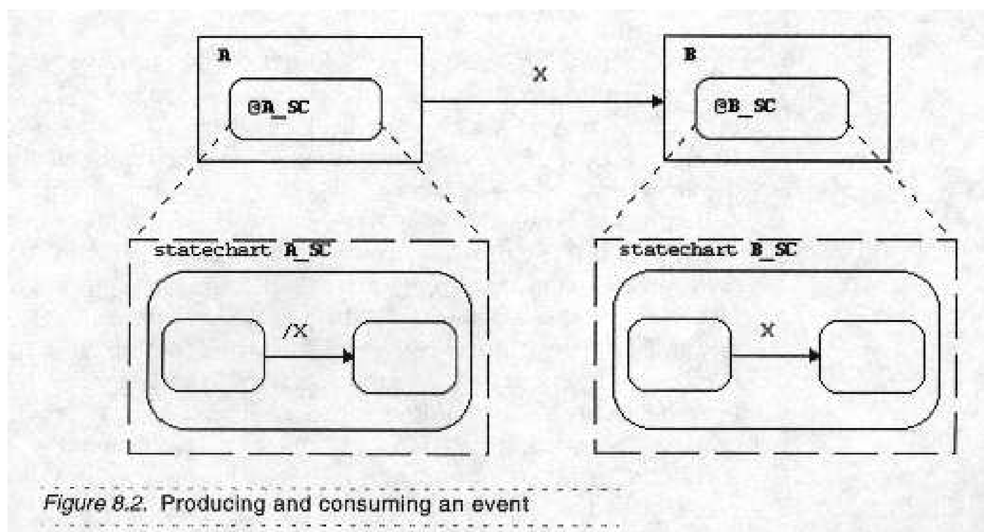


Figure 8.2. Producing and consuming an event

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)).

3.3 Examples of Communication Control

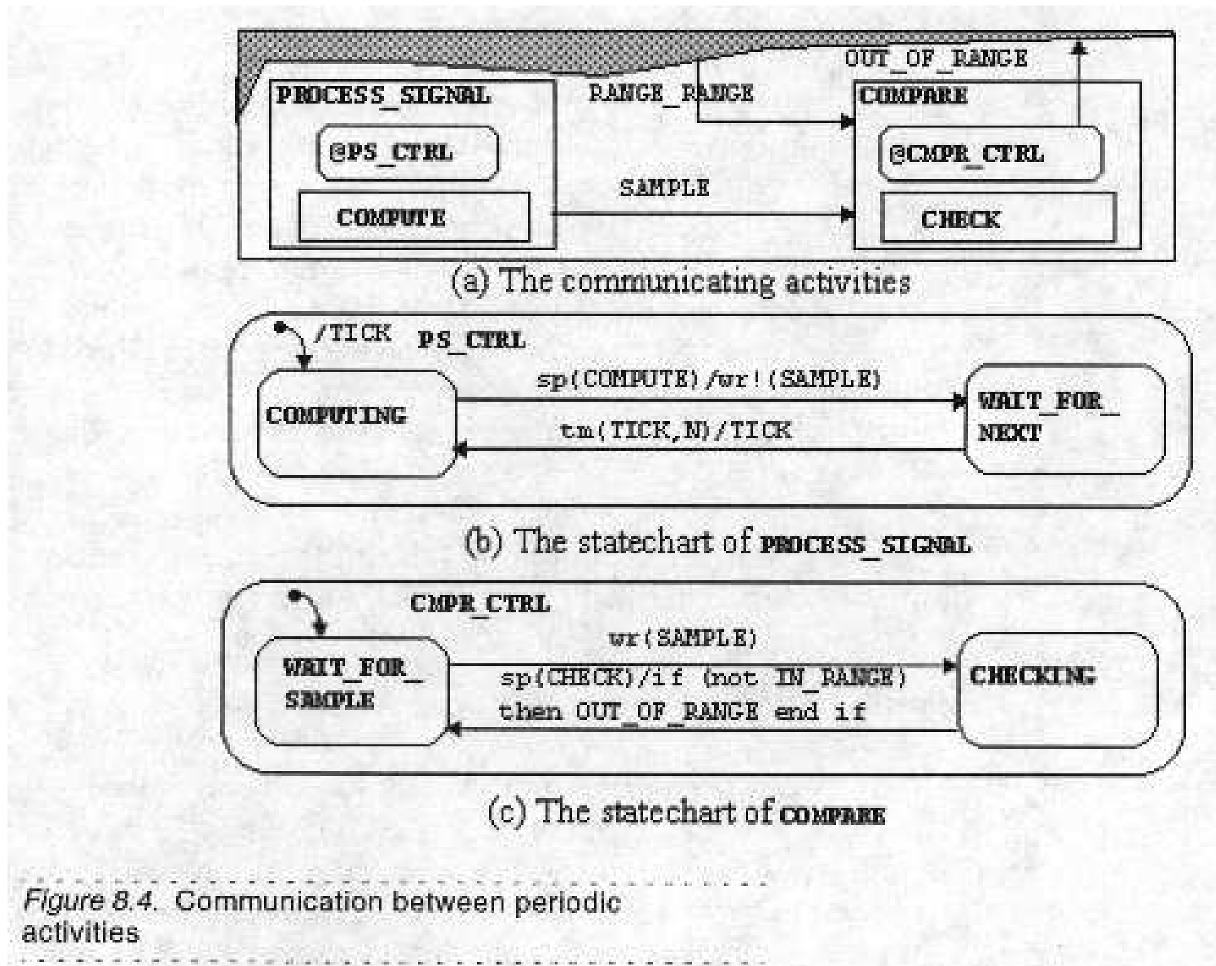
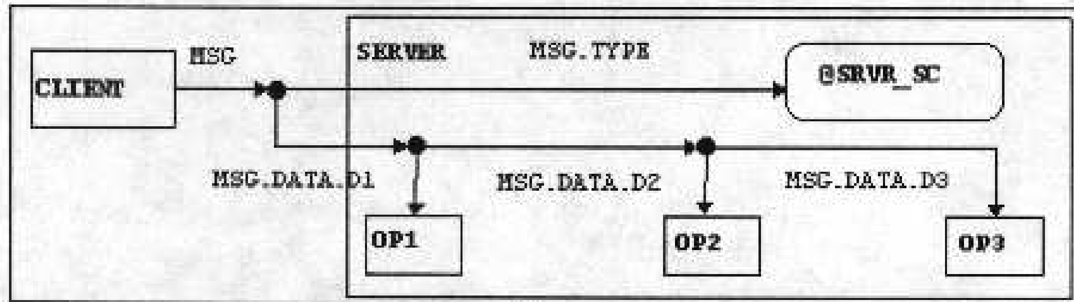
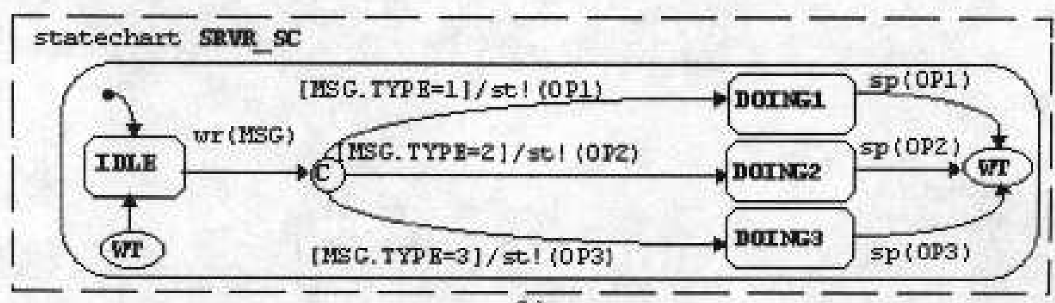


Figure 8.4. Communication between periodic activities

Message Passing



(a)



(b)

Figure 8.5. Server responding to three service requests

3.4 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

3.4.1 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

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

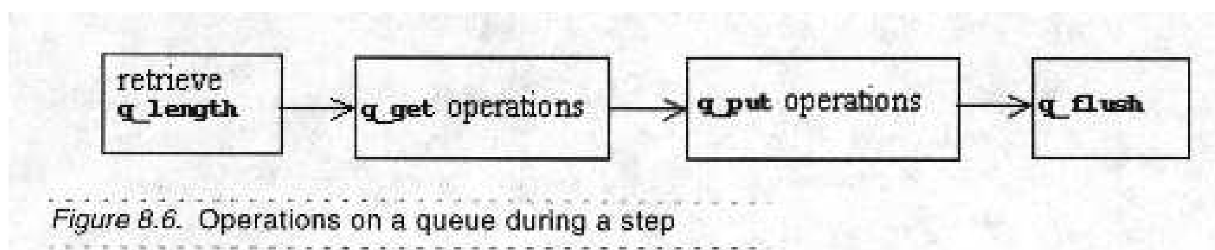


Figure 8.6. Operations on a queue during a step

Combination with Data Stores

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

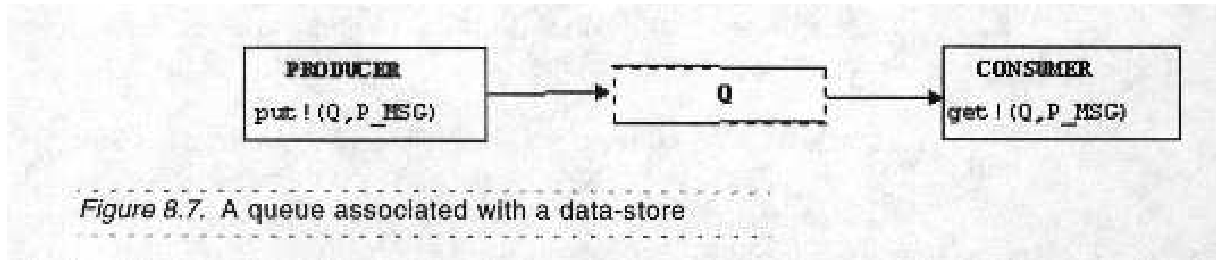


Figure 8.7. A queue associated with a data-store

Example

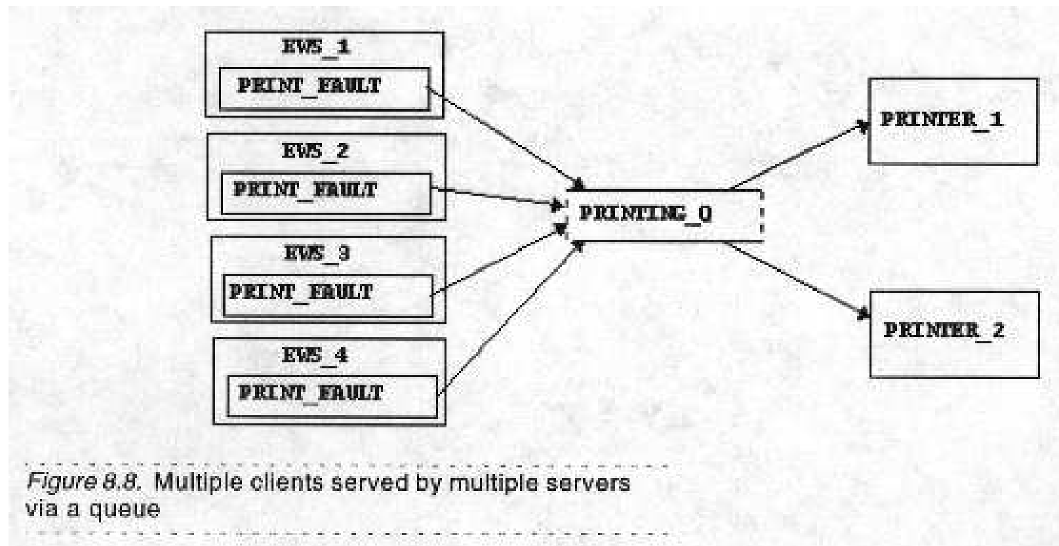
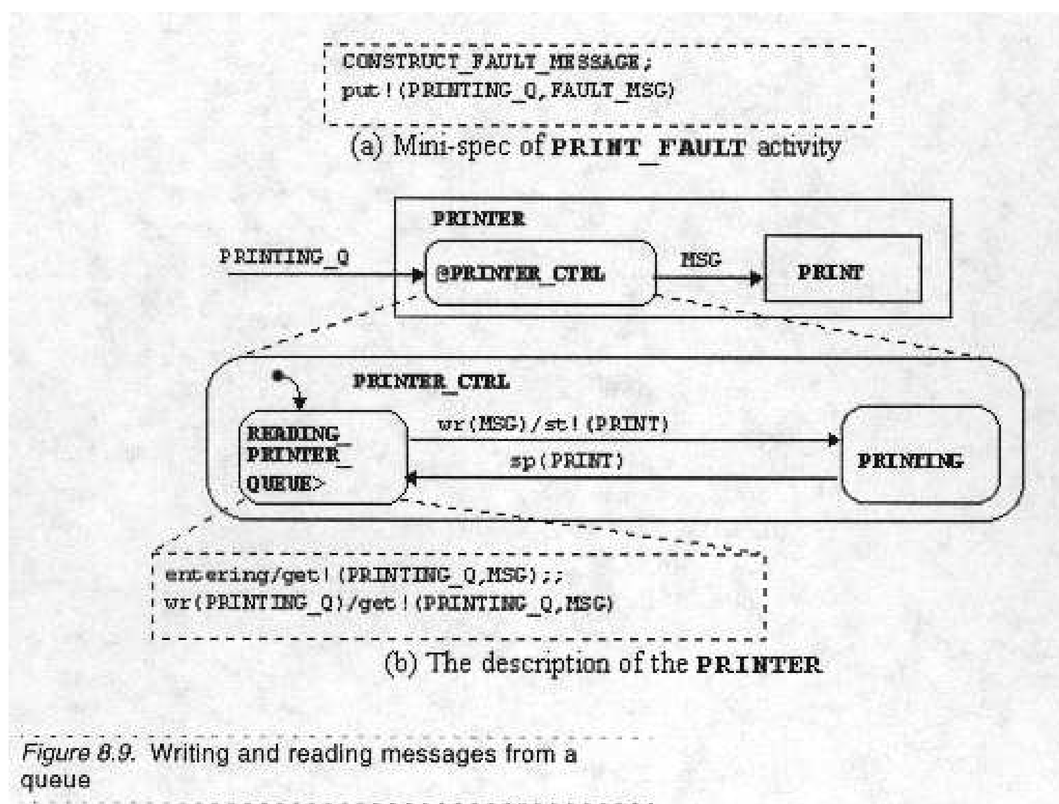


Figure 8.8. Multiple clients served by multiple servers via a queue



(a) Mini-spec of **PRINT_FAULT** activity

(b) The description of the **PRINTER**

Figure 8.9. Writing and reading messages from a queue

5 Statechart Language (cont'd)

5.1 Conditions and Events Related to States

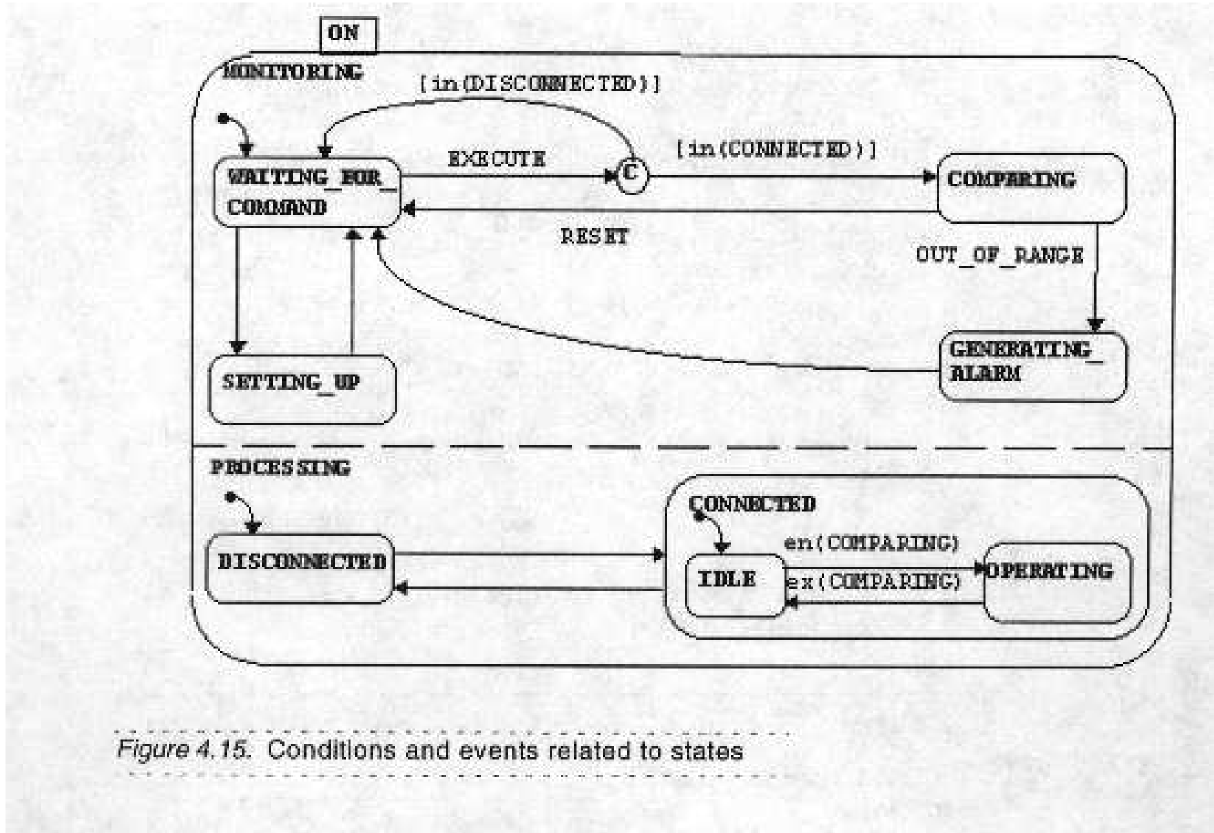


Figure 4.15. Conditions and events related to states

5.2 Connectors

5.2.1 Condition Connector

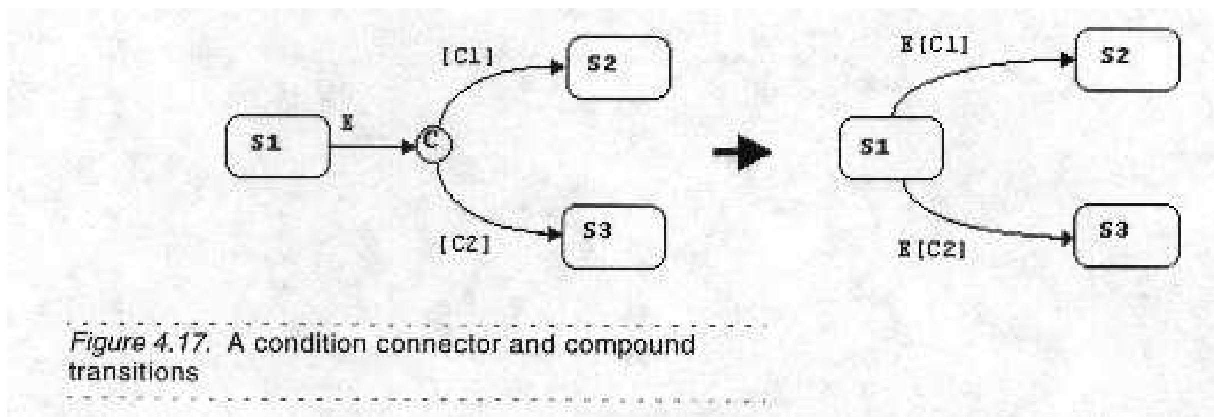


Figure 4.17. A condition connector and compound transitions

5.2.2 Switch Connector

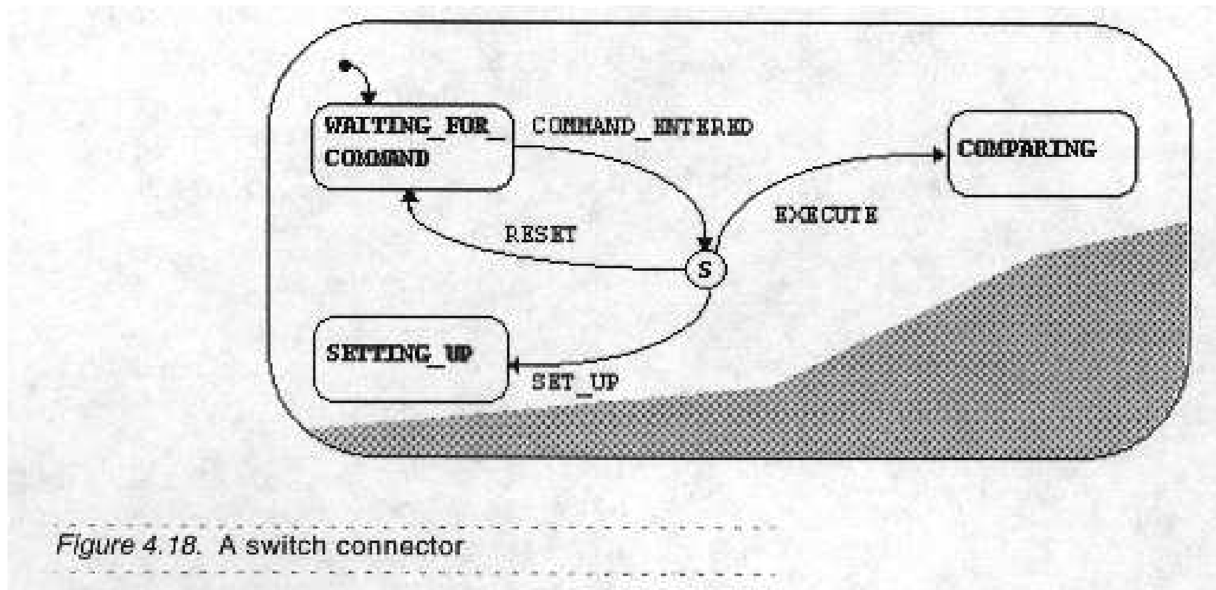


Figure 4.18. A switch connector

5.2.3 Junction Connector

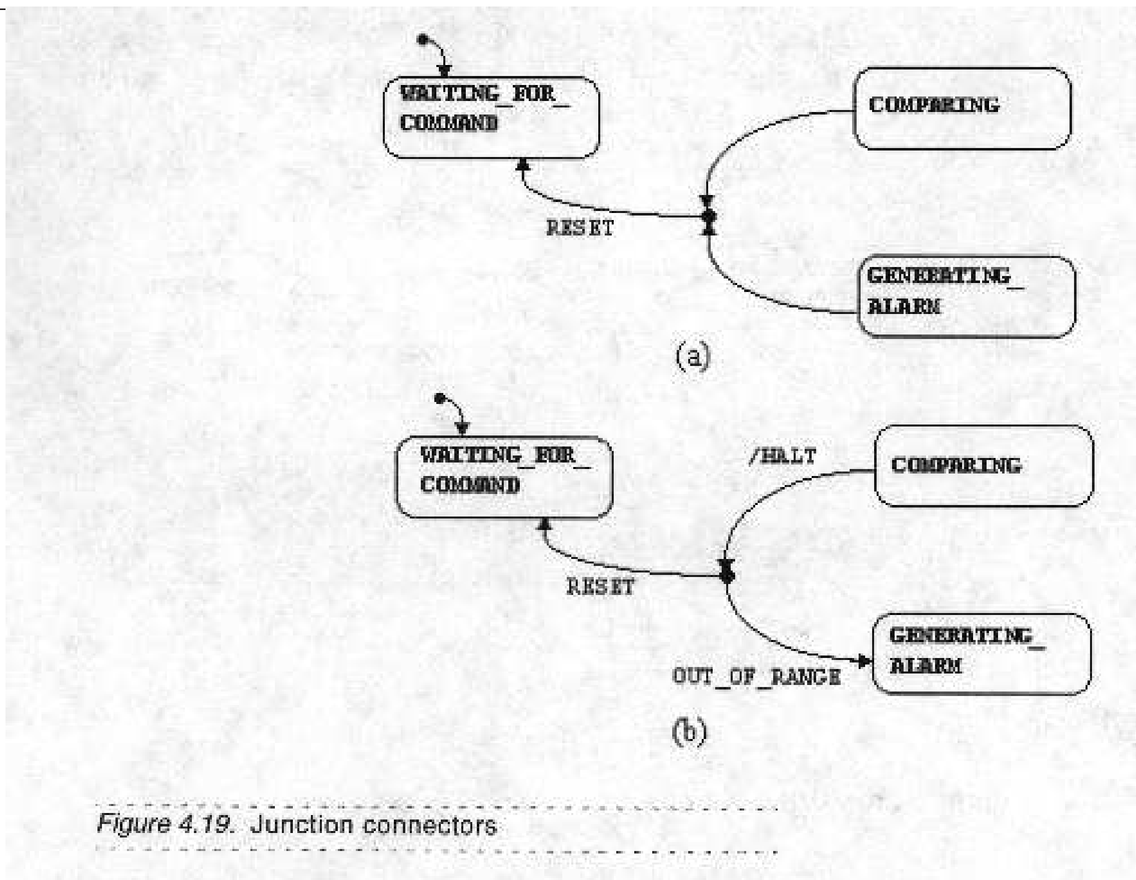


Figure 4.19. Junction connectors

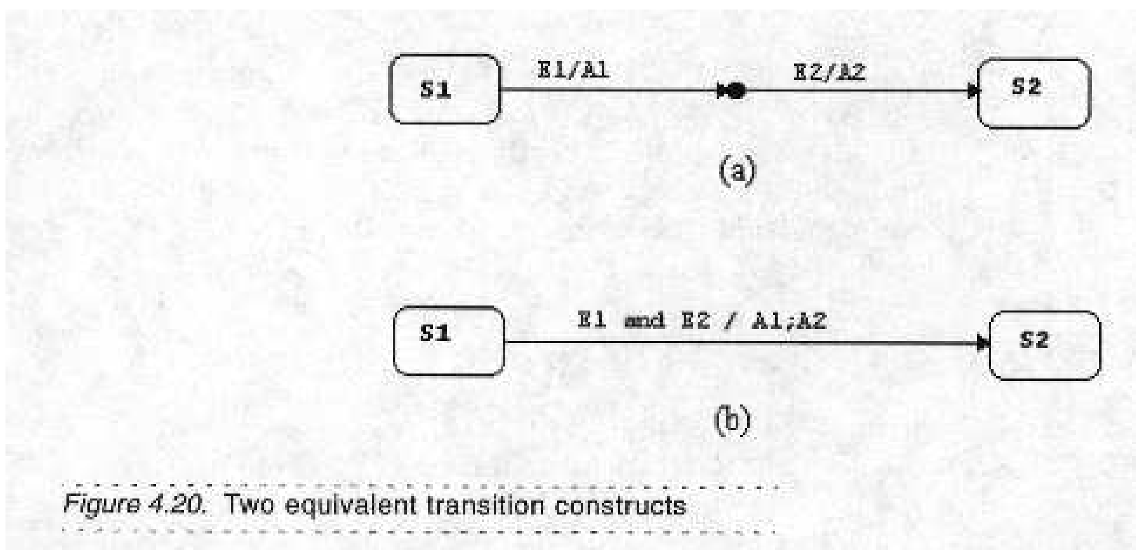


Figure 4.20. Two equivalent transition constructs

5.2.4 Diagram Connector

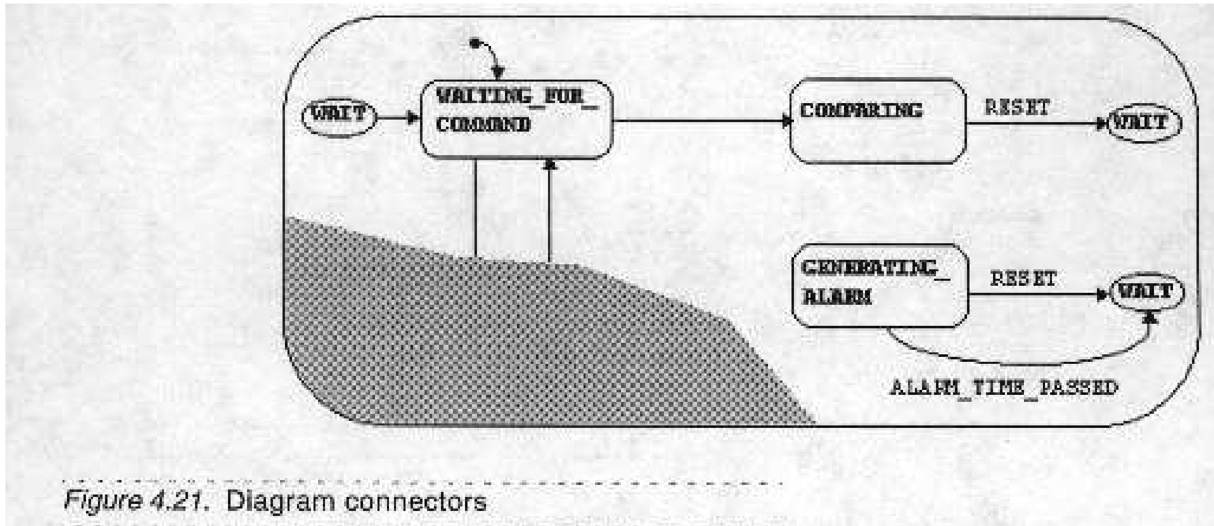


Figure 4.21. Diagram connectors

5.3 Transitions to and from And-States

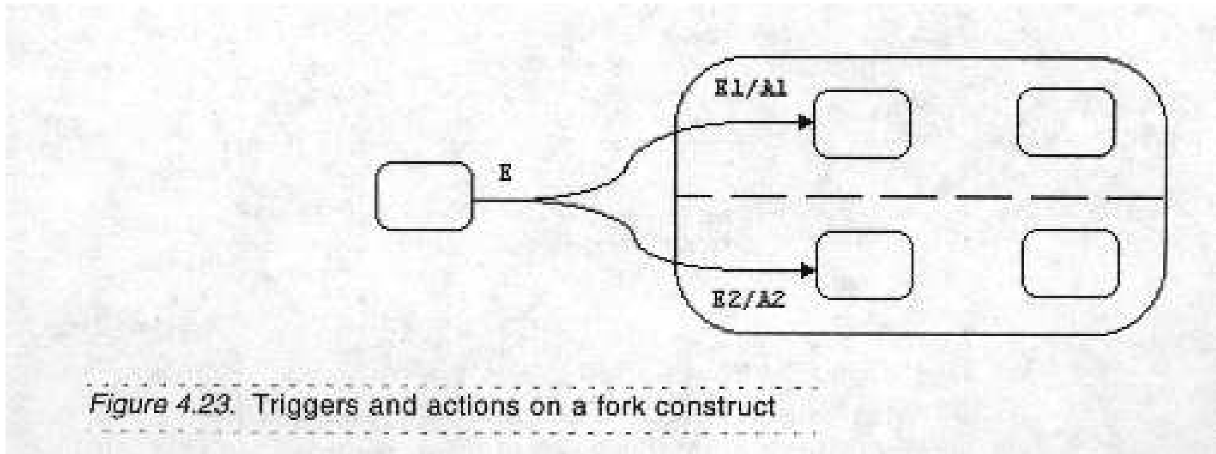


Figure 4.23. Triggers and actions on a fork construct

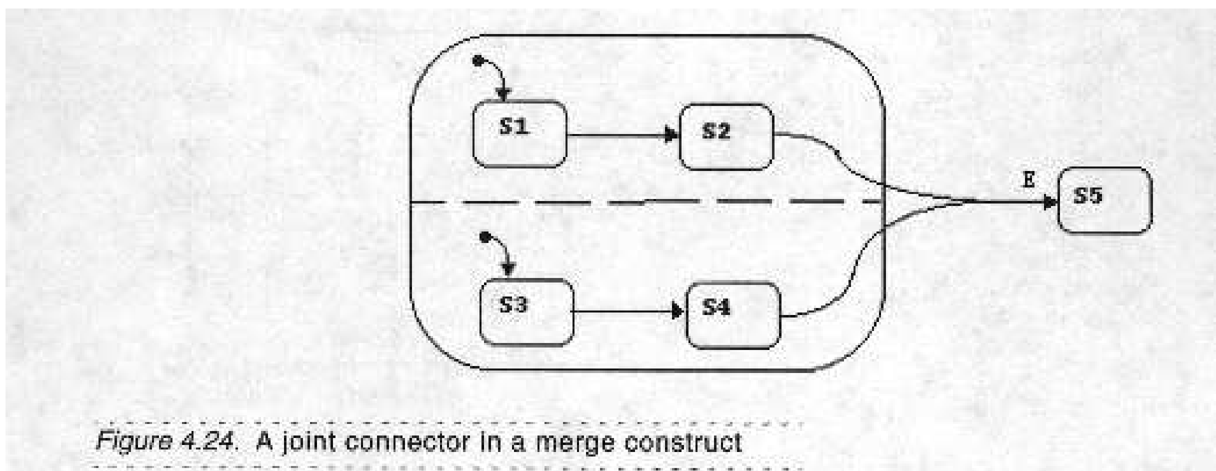


Figure 4.24. A joint connector in a merge construct

Asymmetric Cases

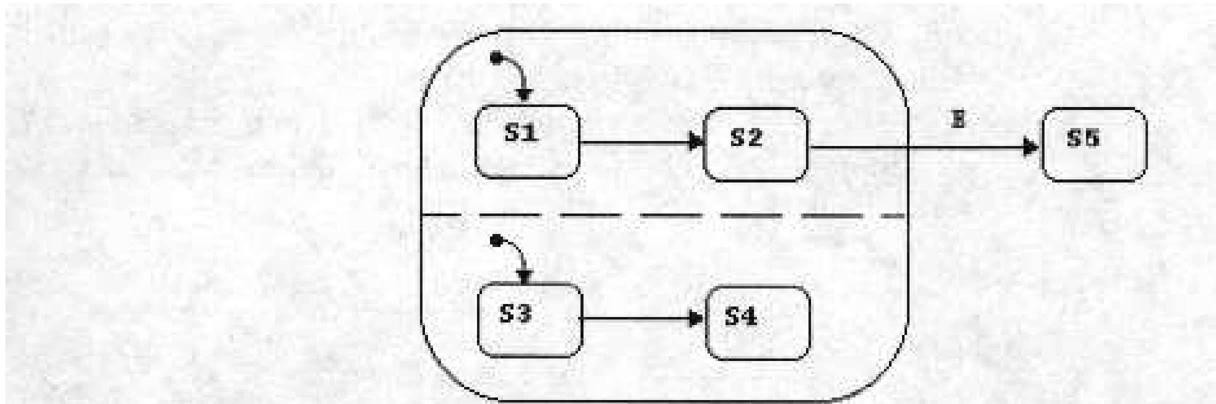


Figure 4.25. A transition from an and-state

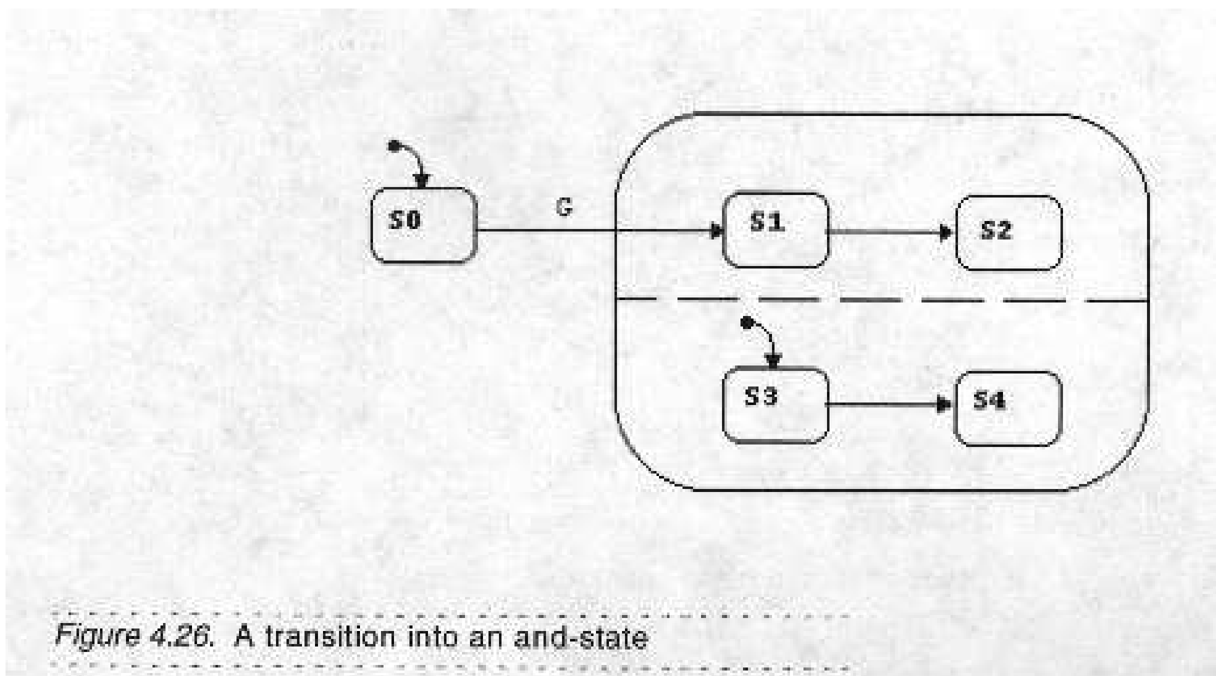


Figure 4.26. A transition into an and-state

Session V

Module-Charts

Abstract: 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.

Literature: Chapters 9 and 10 of "Modeling Reactive Systems with Statecharts", by David Harel and Michal Politi. McGraw-Hill, 1998.

6 Structural Description: High-Level Design

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.

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.

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.

6.1 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**

- 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:

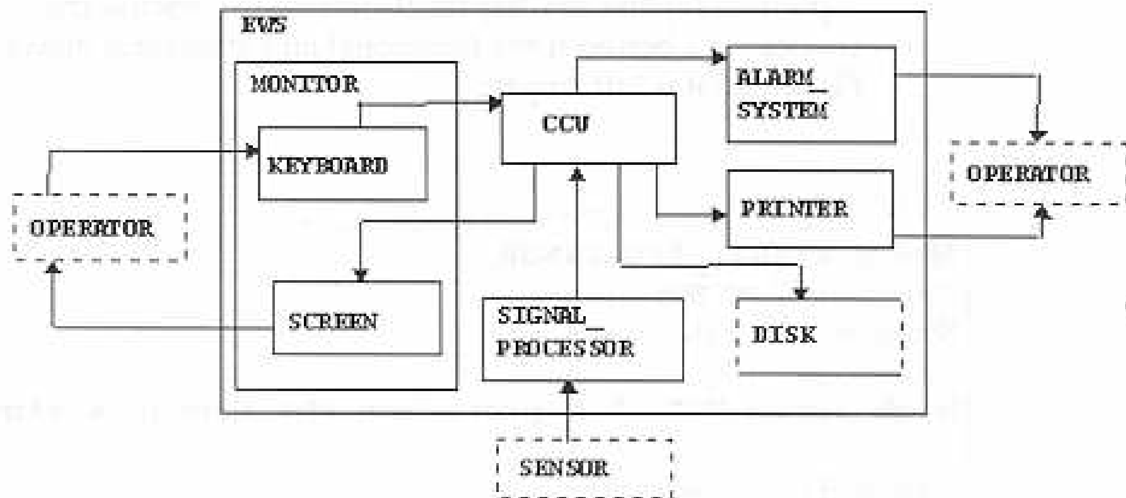


Figure 9.1. Structural decomposition of the EWS

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.

Figure 9.2. A Data Dictionary entry of a module

6.2 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.

A flow-line may denote information flowing between modules:

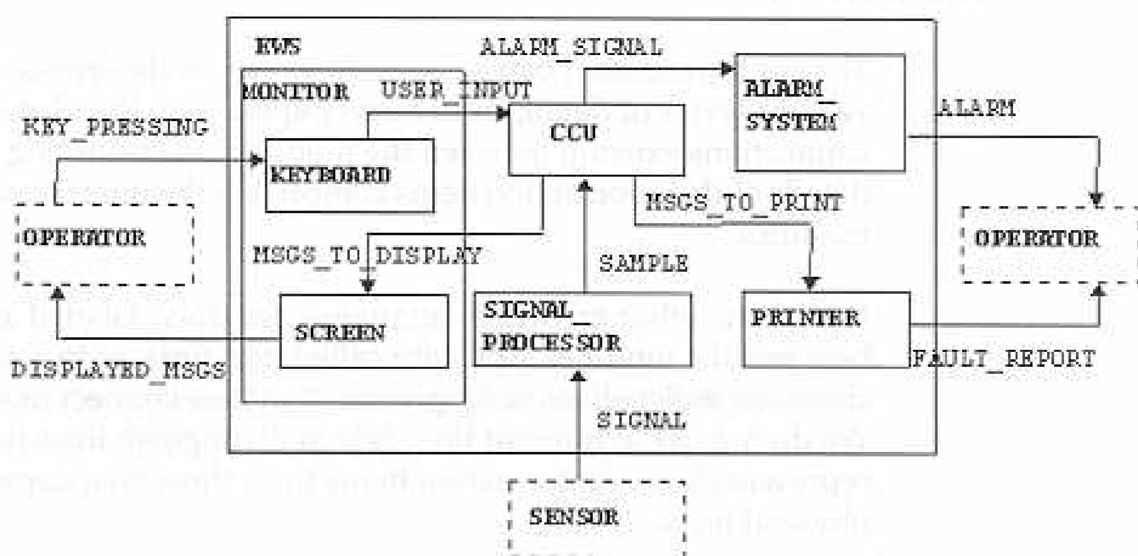


Figure 9.3. Flow of information among modules

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:

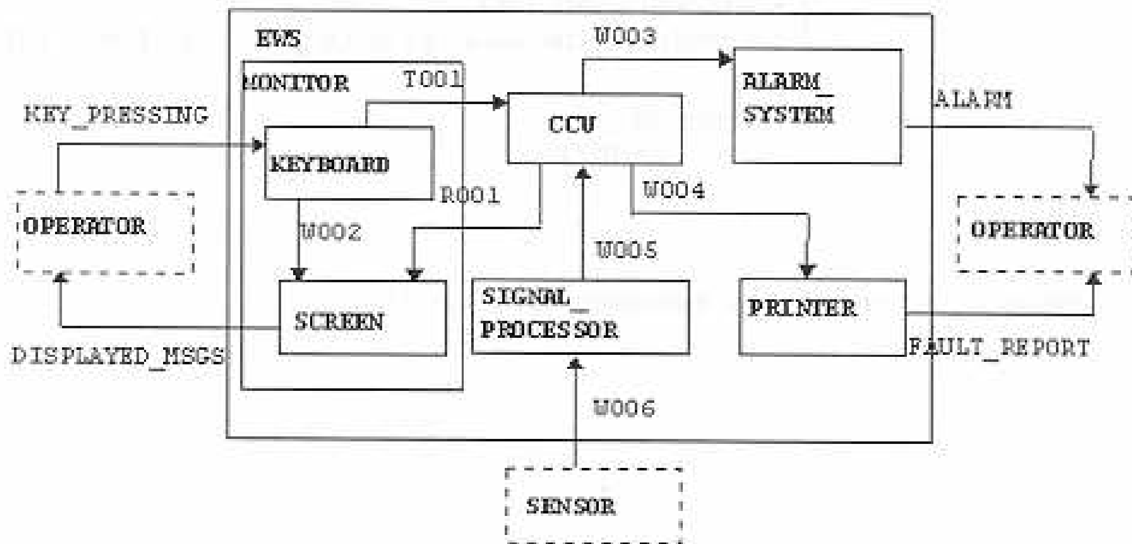
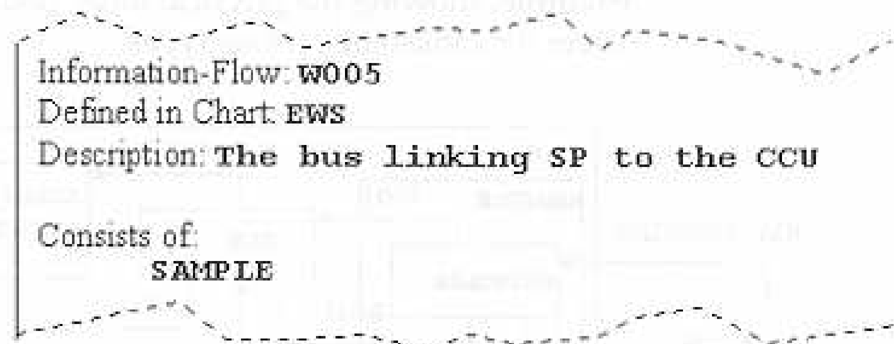


Figure 9.4. Physical links among modules



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

Consists of:
SAMPLE

Figure 9.5. Information-flow describing a physical link

6.3 Connectors and Compound Flow-Lines

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

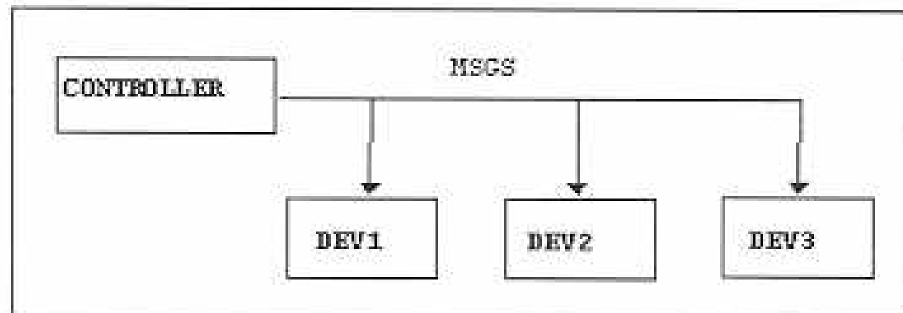


Figure 9.6. Communication link to several devices

7 Connections Between the Functional and Structural Views

- 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.

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 Activities**

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

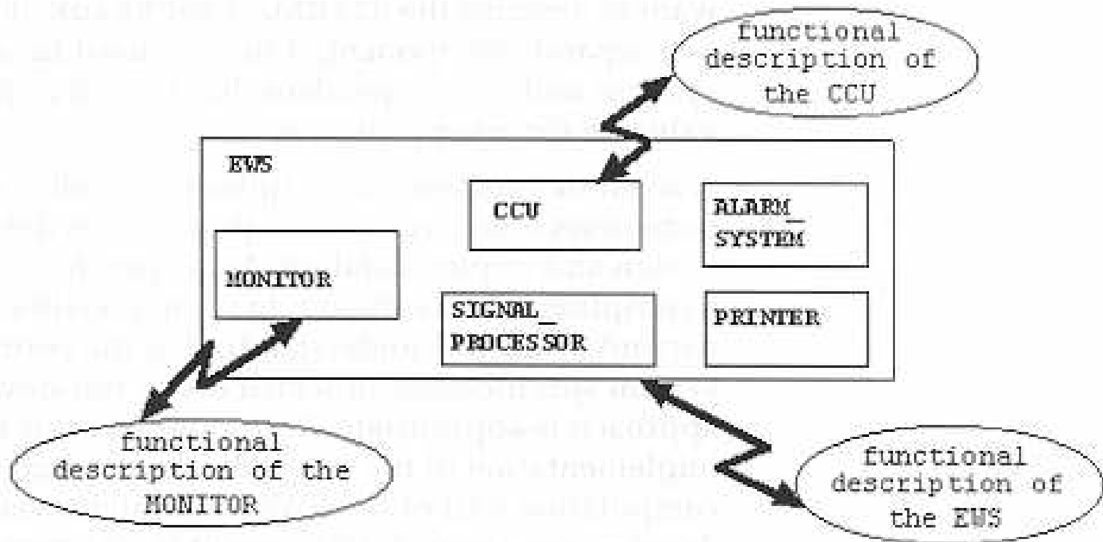


Figure 10.1. Functional descriptions attached to different modules

7.1 Activity-chart Describing a Module

The activity-chart EWS_ACTIVITIES

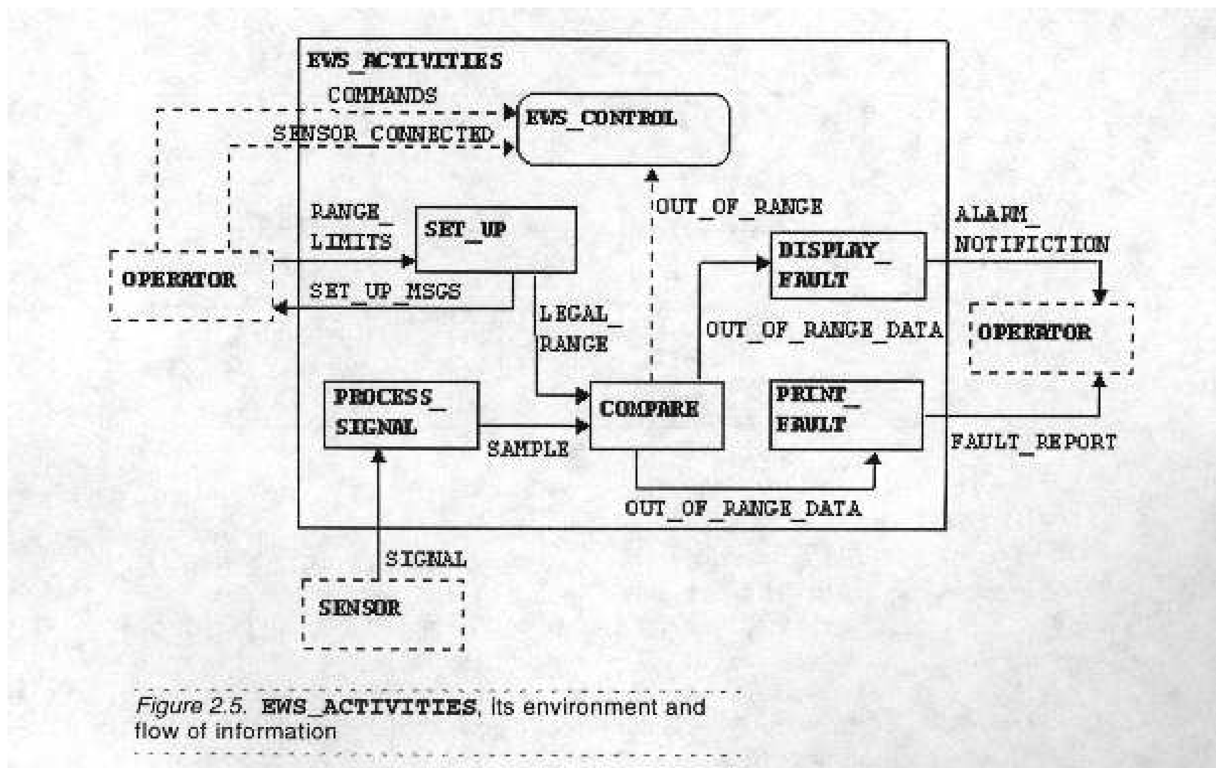


Figure 2.5. EWS_ACTIVITIES, Its environment and flow of information

describes the functionality of the module EWS

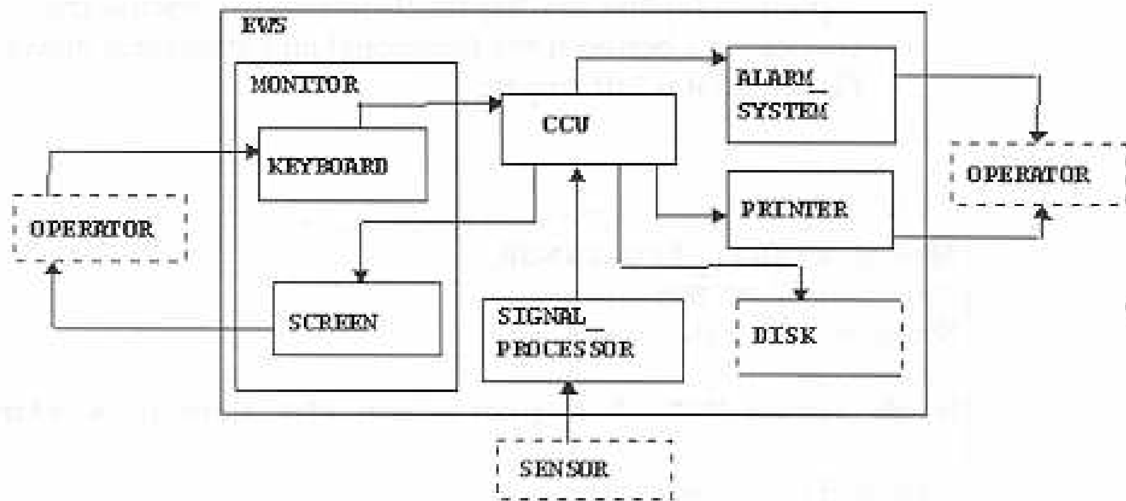


Figure 9.1. Structural decomposition of the EWS

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:

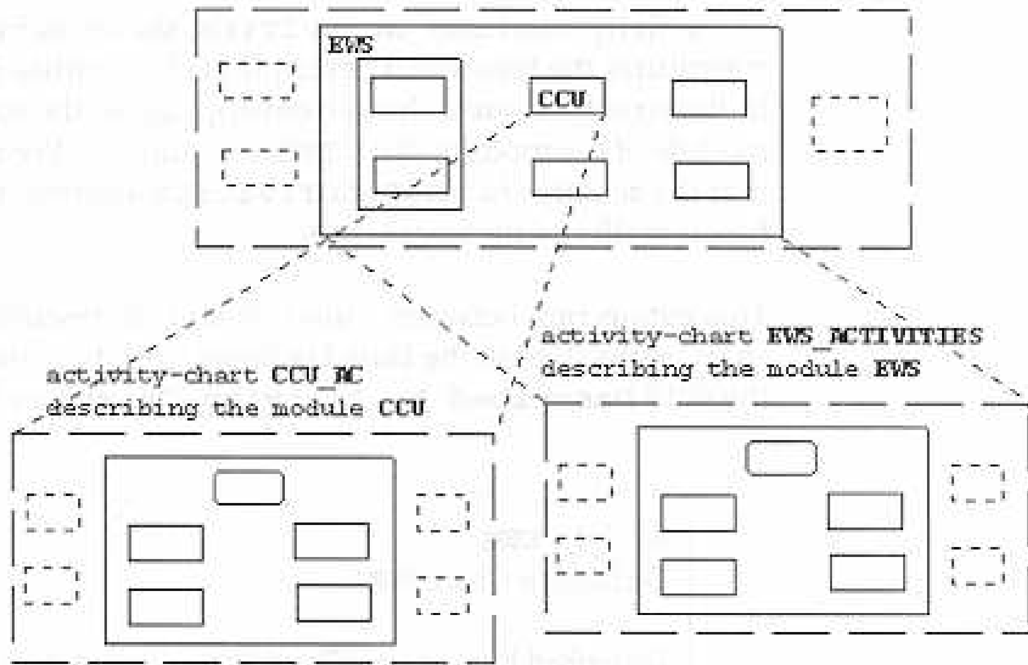


Figure 10.3. Activity-charts describing modules

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

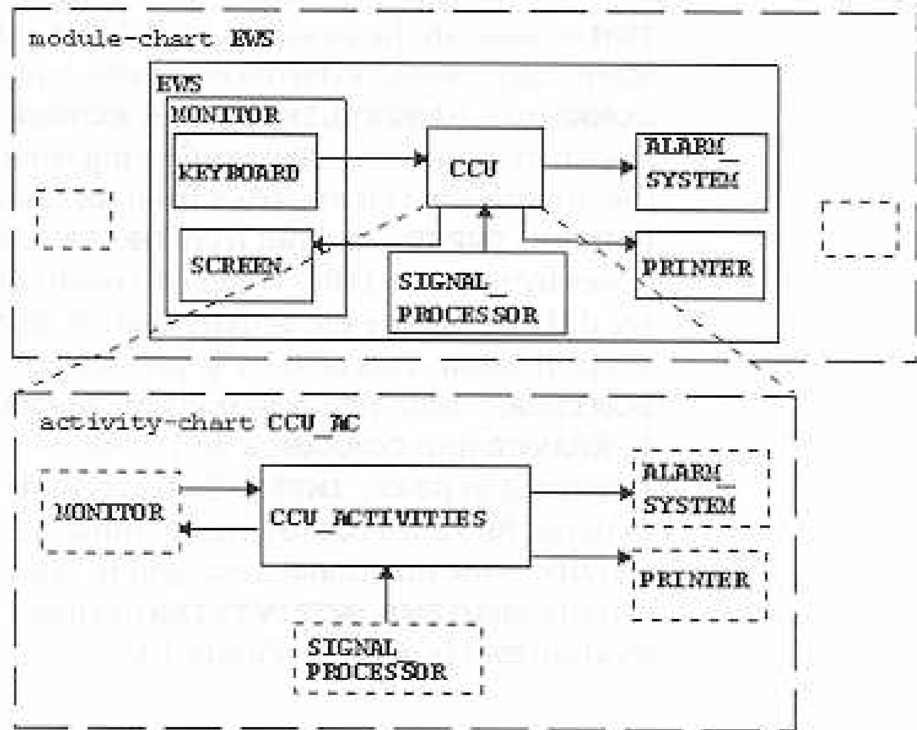


Figure 10.4. External activities corresponding to modules

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:

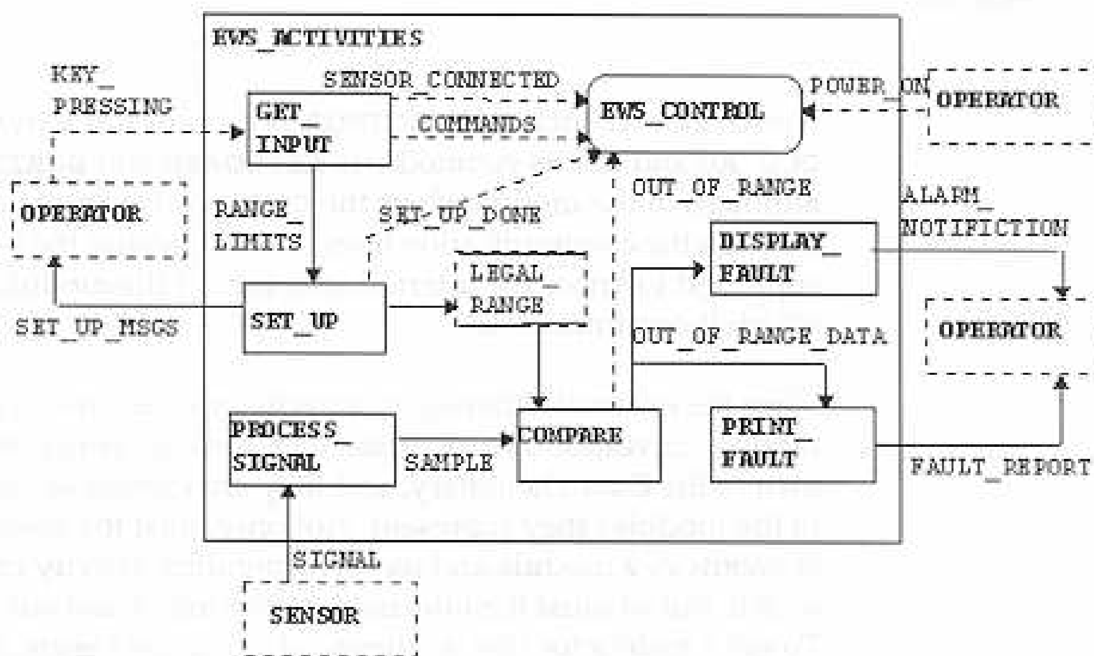


Figure 10.5. Revised activity-chart describing the EWS module

7.2 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:

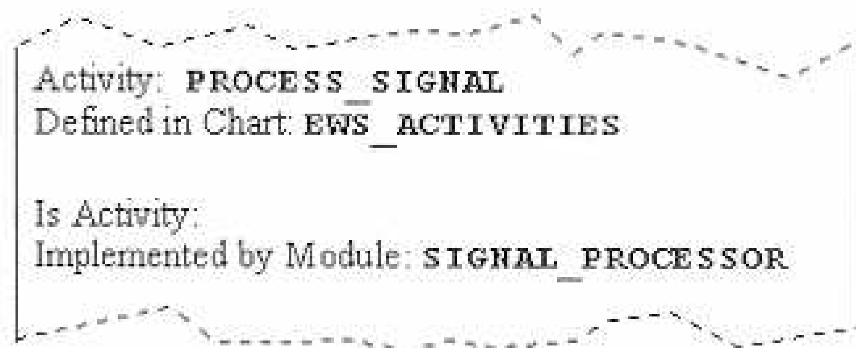


Figure 10.6. An activity implemented by a module

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:

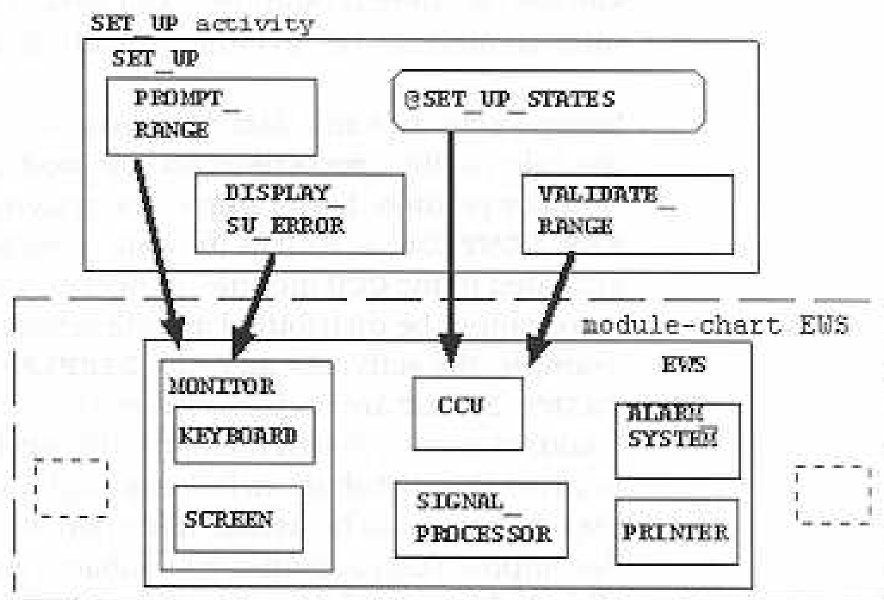


Figure 10.7. Allocation of subactivities of **SET_UP** to modules

7.3 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:

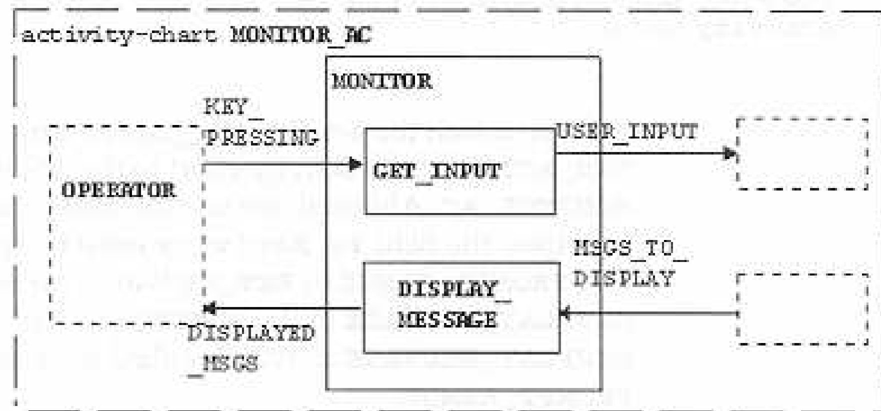


Figure 10.8. Activity-chart of **MONITOR**

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

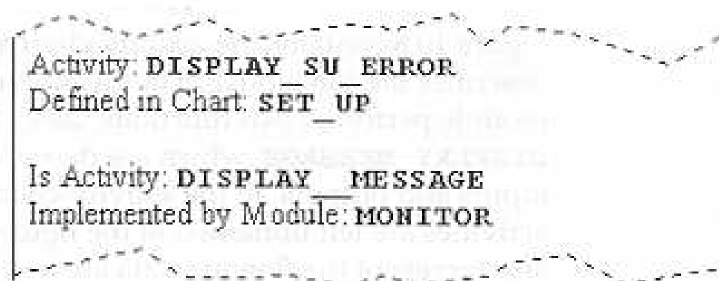


Figure 10.9. Mapping of activities by the **is activity** relation

Session VI

Semantics of Statecharts

Abstract: We discuss the central concepts and decisions for various possible semantics for Statecharts (and the "real" implemented one).

Literature: Dissertation Kees Huizing: "Semantics of reactive systems: comparison and full abstraction", Eindhoven University of Technology, 1991.

In particular the following pages are relevant:

- "Everything you always wanted to know about Statecharts", Huizing and de Roever.
- "On the semantics of reactive systems", Huizing and Gerth.

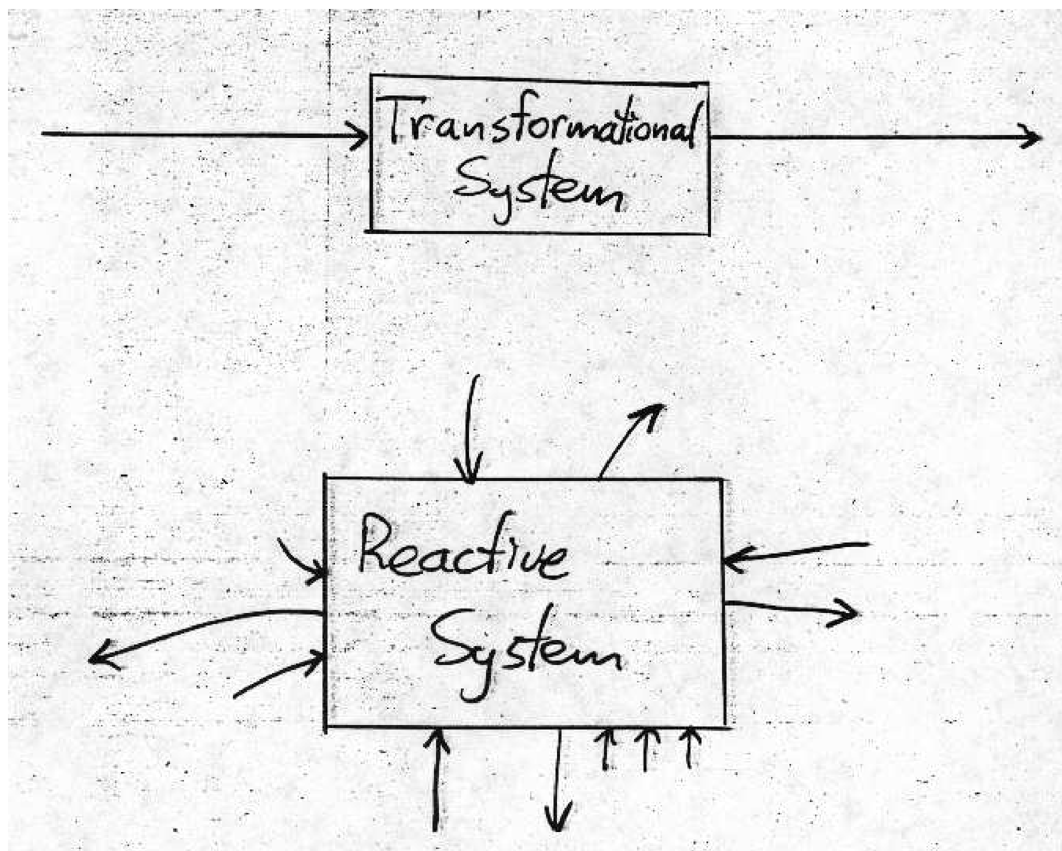
And:

Chapter 6 of "Modeling Reactive Systems with Statecharts", by David Harel and Michal Politi. McGraw-Hill, 1998.

8 Semantics of Statecharts

8.1 Summary of previously discussed material (cfr. first lesson)

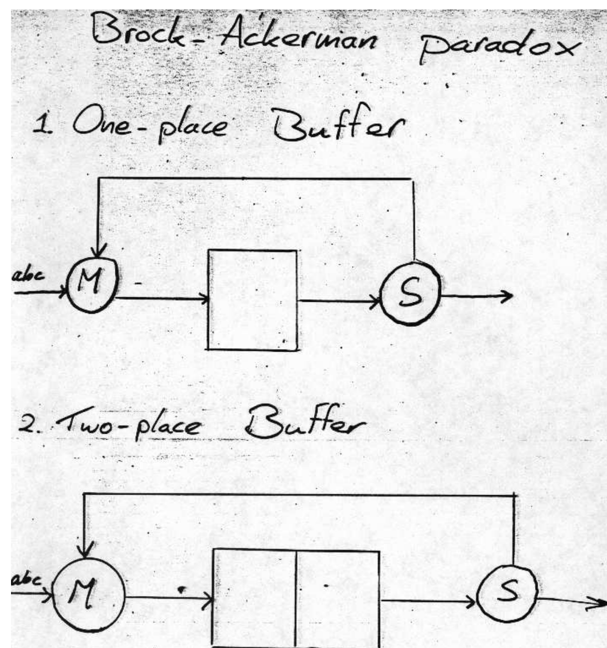
- There is a fundamental dichotomy between **transformational systems** described by the relation between initial and corresponding final states, i.e., their input/output behavior, and
- **Reactive systems**, whose only purpose is to maintain an ongoing relationship with their environment.



Brock-Ackermann Paradox

The Brock-Ackermann paradox explains why reactive systems cannot be characterized by a function mapping sequences of inputs to sequences of outputs.

- Consider two systems, a one-place buffer and a two-place buffer. If you consider these transformationally, they display the same initial-final state behavior.



in: $abc\ldots$
out: 1. e.g. $abac\ldots$
2. e.g. $abca\ldots$
not $abac\ldots$

But if the output of these systems is fed back, and merged with their input they behave differently. (See transparency)

- What's needed to characterize a reactive system is **recording the relative order of inputs and outputs**, i.e., the way they are interleaved.

Central Decisions for a Statecharts Semantics

- Semantics of reactive systems is state-based

- Observations are sequences of pairs of inputs I and corresponding outputs O, i.e., of pairs of the form (I, O).
  In practice a reactive system is therefore described by sequences of the following form:

$$S_1 \Longrightarrow_{I_1}^{O_1} S_2 \Longrightarrow_{I_2}^{O_2} S_3 \Longrightarrow_{I_3}^{O_3} \Longrightarrow \cdots$$

- Transitions don't take time, time is spent in states.
  This has a simple reason: the reaction of a reactive system to environmental inputs should be always well-defined. As a consequence, state-changes shouldn't take time.

Berry's synchrony hypothesis

Reaction time between input (i.e., trigger) and corresponding output (i.e., response) is zero.

Why?

- Recall that individual reaction times are too complicated to handle, abstractly, on the high level of specification Statecharts are aiming at.
- a fixed non-zero reaction time wouldn't allow transition refinement.
- Unspecified reaction times lead to chaos, and is not desired at a high level of abstraction.

$\Longrightarrow$ Only one reaction time satisfies all criteria: zero! For:

- Now transition can always be refined
- specific
- deterministic

Detailed Argumentation from Lesson I

Possibility 1 : Specify a concrete amount of time for each situation. This forces us to quantify time right from the beginning. **Clumsy**, and not appropriate at this stage of specification where one is only interested in the relative order and coincidence of events.

Possibility 2 : Fix reaction time between trigger **a** and corresponding action **a** within **e/a** (the label of a transition) upon 1 time unit.

Doesn't work: Upon refining **question/answer** to a **question/consult** and a **consult/answer** transition, there's a change of time, which may have far reaching effects (because of **tm(n)**-events, e.g.)

$\Longrightarrow$

A fixed execution time for syntactic entities (transitions, statements, etc.) is not **flexible** enough.

Possibility 3 : Leave things open: say only that execution of a reaction takes some positive amount of time, and see at a later stage (closer to the actual implementation) how much time things take.

Clumsy, introduces far too much nondeterminism.

Reaction time of a system (2)

Summary : We want the execution time associated to reactions to have following properties:

- It should be accurate, but not depending on the actual implementation.
- It should be as short as possible, to avoid artificial delays.
- It should be abstract in the sense that the timing behavior must be orthogonal to the functional behavior.

Only choice that meets all wishes is **zero reaction time**.

As a result all objections raised w.r.t. the possibilities mentioned on the previous page are met!

- Now, for instance, upon refining transition **question/answer** from previous page into two transitions, the reaction time of this refinement is the same as that of the original transition.
- Objection 3 on the previous transparency is resolved, too.
- Finally, also objection 1 (on previous transparency) is met, because $0 + 0 = 0$!

Berry's Synchrony Hypothesis

- Is Berry's synchrony hypothesis implementable?
  Yes, if the input frequency is low w.r.t. the time required for computing response.
- However, this hypothesis leads to a number of counterintuitive consequences, if carried through.
  Careful: the following example does not describe the Statecharts semantics as implemented in Statemate.

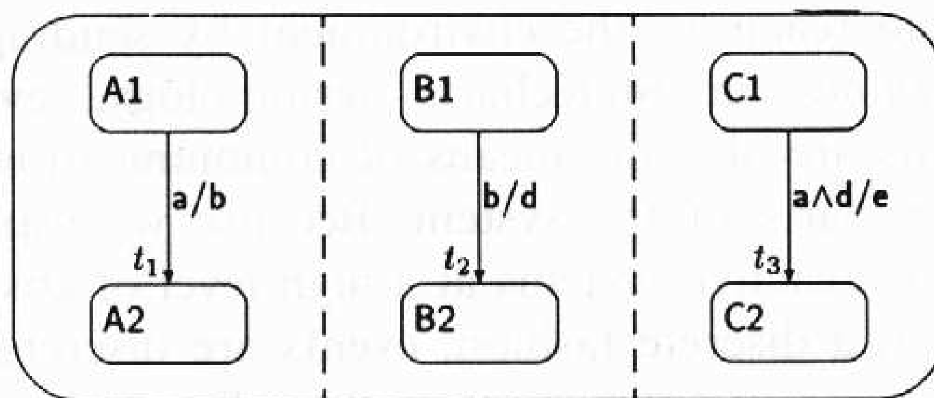


Fig. 6.

A $\wedge$ D is a generated trigger, since we assume the reaction time to be zero. A consequence is that transition t_3 is taken!!

Combination with Negation

The synchrony hypothesis leads to problems if combined with the possibility of checking the absence of signals (the latter is customary in the synchronous world, and a possibility not offered in the asynchronous world):

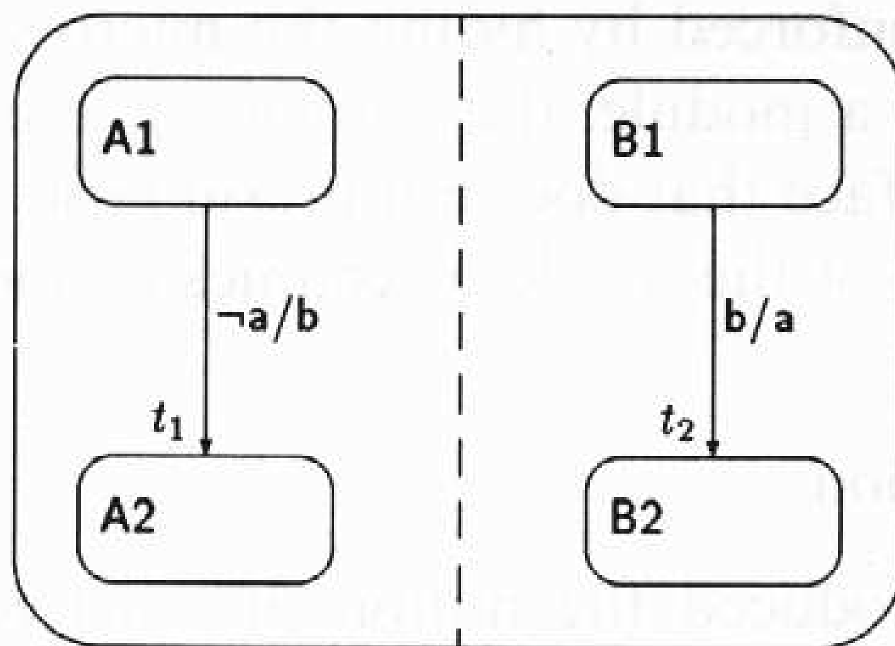


Fig. 8.

If **a** is absent, i.e., ¬ **a** holds as condition, transition t_1 is taken, i.e., **b** is generated, and hence t_2, i.e., **b/a** is taken, generating **a** **within the same time unit**, i.e., in zero time, hence transition t_1 should not be taken.

This is called the “Grandfather paradox”.

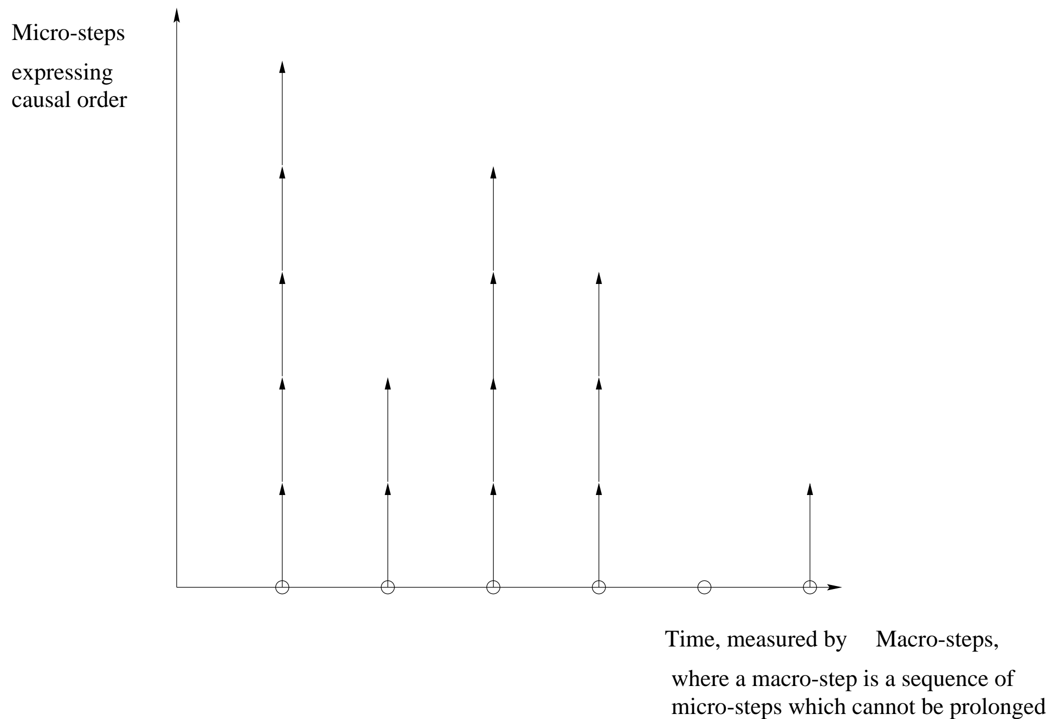
It’s solution is to **order** event occurrences causally, with **later events not influencing earlier events**:

$$\neg a \leq b \leq a$$

> Note here: this causal order has nothing to do with the passage of time; it merely refers to causal chains **within** one time step.

The new semantics

- This leads to a semantics of the following form:



- Macro-steps are observable steps $\overset{O}{\underset{I}{\Longrightarrow}}$
- Each macro-step is a sequence of micro-steps, that are ordered causally; one micro-step can never influence previous micro-steps.
- In Statecharts as implemented by Statemate causality is trivially obtained because in Statemate events generated in one step are only available in the next step, and only for that one. I.e., there is no causality within one step.

Problems with this new semantics

- The problem with macro-steps is that they lead to a **globally inconsistent** semantics, i.e., transitions are taken in one macro-step which aren't generated globally.

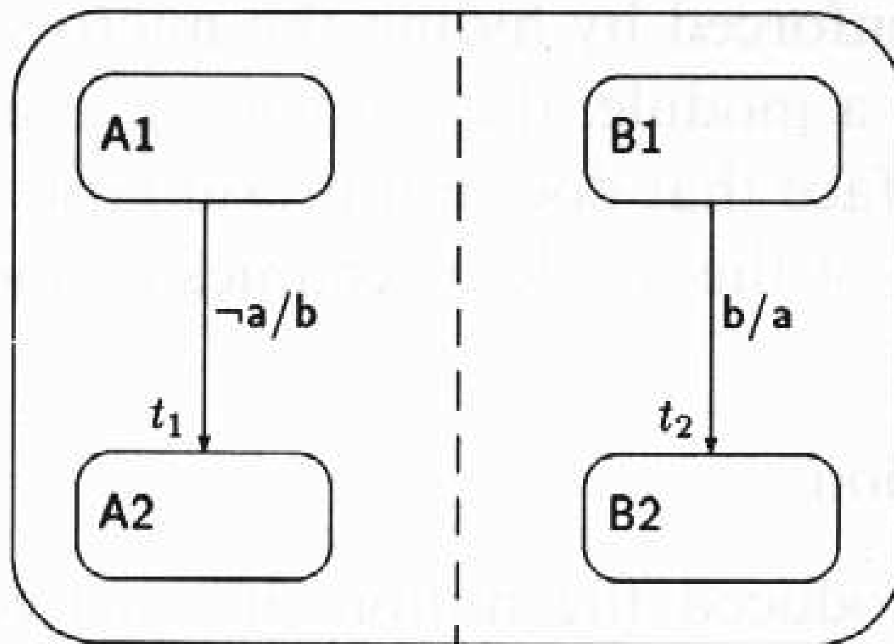


Fig. 8.

$$S_1 \Longrightarrow^{b}_{\emptyset} S_2 \Longrightarrow^{a}_{b} S_3$$

Here absence of triggers generates presence of triggers, which violates their absence within the same step (not globally consistent).

- These considerations lead to the fundamental question:

  Is a semantics for such languages possible which satisfies all "reasonable" assumptions? I.e., which is both good for program development and for program composition? The answer is **NO**.

- This is a serious problem. As it turns out, the semantics with macro-steps indicating passage of time, and refined by causally ordered micro-steps is a basis for a compositional semantics for Statecharts in which the semantics of a construct is a function of the semantics of its parts. But this semantics turns out to be too difficult to handle for the engineers of I-Logix, and of Israeli Aircraft Industries, its main customer for the Statemate system.

- Hence looking for a "best" semantics makes a lot of sense. What our theorem below says is that, in a certain sense, there is no best semantics. However, it does leave some room for the search for ever better semantics!!

There is no "best" semantics for Statecharts

Let's list a couple of desirable properties of such a semantics:

Responsiveness: Reactions are simultaneous with their triggers — this facilitates refinement of transitions from a high to a lower level.

Causality: Without a causal order of the micro-steps inside a macro-step, charts s.a.:

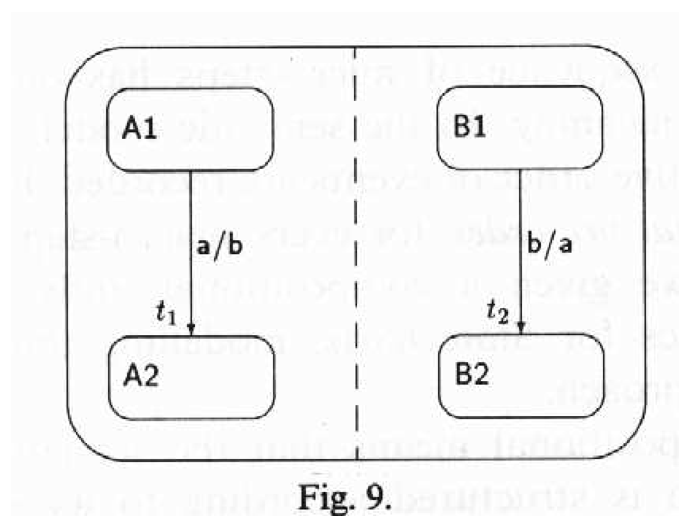


Fig. 9.

would trigger each other, which makes no causal sense. Such charts are excluded imposing causality.

Modularity: Modules can be composed on the basics of their macro-steps, i.e., the external interface of a (parallel) composition of modules is of the same nature as their mutual interface w.r.t. each other. (This is inspired by a paper by Pnueli and Shalev)

Impossibility of a Semantics being Causal, modular, and responsive

Modularity, causality, and responsiveness can be mathematically expressed; the **impossibility** of all three being satisfied simultaneously becomes a theorem, proved in the paper by Huizing and Gerth.

However, also intuitively this is clear:

- Causality and responsiveness leads to

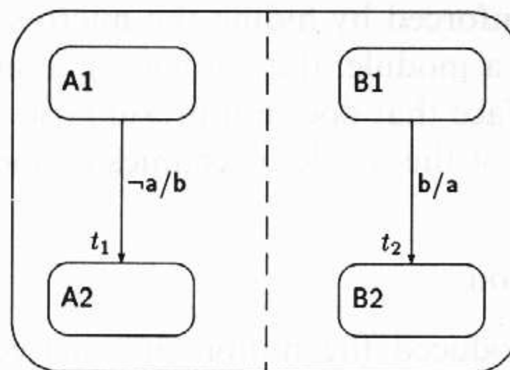


Fig. 8.

  examples in which both a and its absence $\neg a$ occur within the same macro-step $\Longrightarrow$ no global consistency $\Longrightarrow$ no modularity

- Modularity and responsiveness imply there exists no satisfactory semantics for the example above. This choice is made in the synchronous language ESTEREL, in which

examples as the one above are excluded on syntactic grounds by a compiler.

8.2 Classification of possible semantics for Statecharts

Next we list a few possible semantics for Statecharts, semantics A – E, of which E is closest to the one actually implemented in Statemate, and discuss the anomalies allowed by them (including those of the implemented semantics of Statecharts).

Semantics A

Events generated as a reaction to some input can only be sensed in the step following that input. (This is a choice made in the implemented semantics of Statecharts.)

Anomaly: no simultaneity of action and reaction, i.e., no responsiveness.

In semantics A the trigger $a \wedge c$ will not occur:

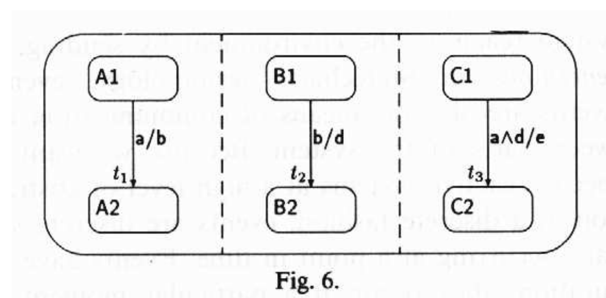


Fig. 6.

This example makes clear that in semantics A the moment of generation of an event is too important — a too detailed analysis of charts is required for adopting it.

Semantics B

In order to overcome the problem with semantics A, absence of responsiveness, micro-steps are introduced, with events sensed in the next micro-step.

Then, in the previous example the third transition is taken.

Consider now the trigger $b \wedge \neg c$ for the third transition; the transition is taken, because in the second micro-step, event c is not yet sensed. This example also works for semantics A.

Disadvantage: Semantics B is too subtle to be of any practical use; same objection as to semantics A.

Semantics C

Requires global consistency of **every** micro-step. The reaction of the system to an input should

- not only be enabled by events generated in previous micro-steps
- but also by events generated in the full macro-step.

As a consequence, the $b \wedge \neg c$ transition is not taken.

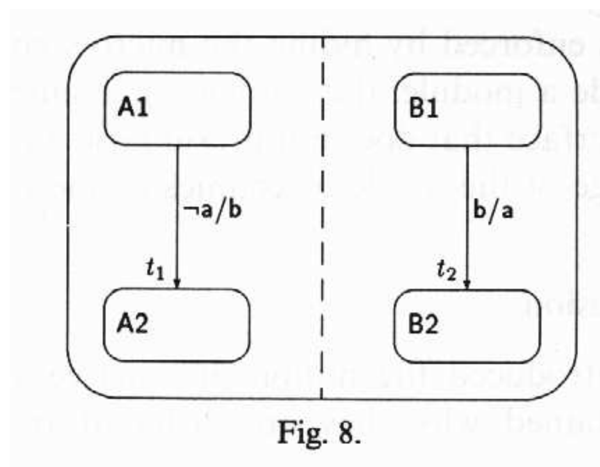


Fig. 8.

This example is excluded in semantics C, leads to contradiction. I.e., syntactical means must be found to exclude it, as done in ESTEREL by a compiler.

This makes a lot of sense, as evidenced by the considerable success of ESTEREL of Gérard Berry.

However, this semantics is not modular. This implies that a modular development of the system is cumbersome, since every developer has to know the detailed micro-behavior of the other processes. Hence, this semantics is appropriate for top-level guys only, and that's what Gérard Berry's crowd consists of.

Semantics D

All events generated during some macro-step considered as if they were present right from the beginning of the macro-step.

Semantics D allows

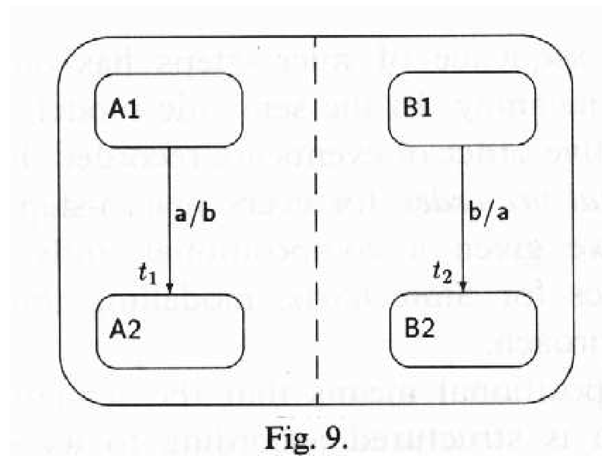


Fig. 9.

to be taken: reactions may trigger themselves. I.e., semantics D is not causal.

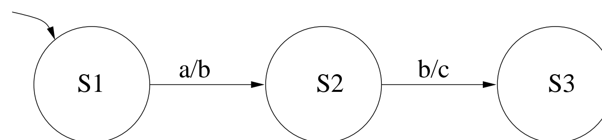
Note: In semantics D, the external world does not generate an a event!

Conclusion: This example should be rejected!

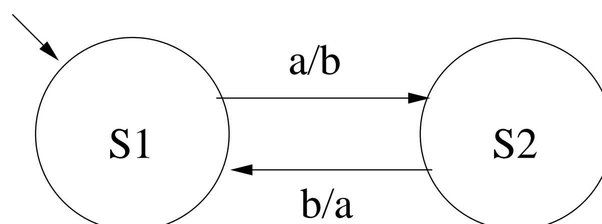
Semantics E

Events are generated at the next step, but no input from the environment is possible **before** the reaction of the system has completely died out.

This semantics is heavily non-modular, since one macro-step may contain several steps of the A semantics. Events remain active only for the duration of such a step, hence, in one macro-step an event can be activated and deactivated several times, thus leading to a much more complex interface between subsystems, than between the system and its environment.



Generation of event a leads the system eventually to state S_3.

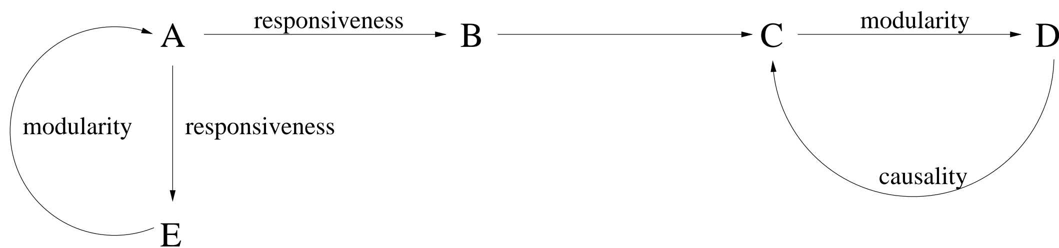


In semantics E, as in the implemented semantics of Statecharts, this example leads to an infinite loop (the so-called: "go repeat" mode): try it out yourself!

Situation

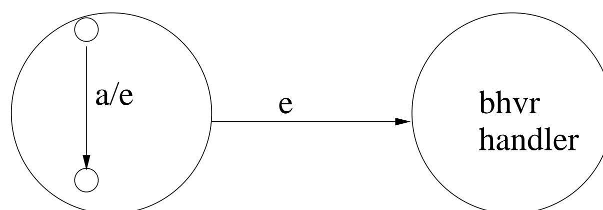
No “best” semantics $\Longrightarrow$ still room for better ones

The situation is summarized in the following figure, showing how each semantics is an attempt to improve on the other one:



What to do? The search is now on for better semantics

1. Several cleaner semantics have been proposed, notably by Florence Maraninchi. She opts for semantics D, in which both charts such as example C and D are excluded, resulting in Argos semantics:

   Generation of event a leads to exit transition e being taken. This is called **non-preemptive interrupts**.

The Argos semantics leads to a cleaner concept of state-hierarchy in which **inter-level transitions are not allowed**. Probably a too heavy investment in their “old” semantics, manyear-wise, prevented I-Logix from adopting the cleaner Argos semantics of Maraninchi in Statemate.

2. Huizing and Gerth propose a compositional semantics in which the causal chains inside a module are hidden from its external behavior. This proposal has not yet caught on.

8.3 Statecharts as Implemented

This leaves us with the semantics of Statecharts as it is implemented in Statemate. Computing that semantics is a fairly involved algorithm, only recently (1996) published in a paper by David Harel and A. Naamad.

Operational semantics

We describe the contents of the system status, and the algorithm for executing a step.

The **status** includes:

- a list of states in which the system currently resides;
- a list of activities that are currently active;
- current values of conditions and data-items;
- a list of regular and derived events that were generated internally in the previous step;
- a list of timeout events and their time for occurrence;
- a list of scheduled actions and their time for execution;
- relevant information on the history of states.

The **input** to the algorithm consists of:

- the current system status;
- a set of external changes that occurred since the last step;
- the current time

The **step** execution algorithm works in three main phases:

1. - calculate the events derived from the external changes and add them to the list of events;
   - perform the scheduled actions whose scheduled time has been exceeded, and calculate their derived events;
   - update the occurrence time of timeout events if their triggering events have occurred;
   - generate the timeout events whose occurrence time has been exceeded;
2. - evaluate the triggers of all relevant transition reactions;
   - prepare a list of all states that will be exited and entered;
   - evaluate the triggers of all static reactions
3. - update the history of states;
   - carry out all computations prescribed by the actions in the list produced in the second phase;
   - carry out all updates called for by the actions
   - update the list of current states.

Synchronous/Asynchronous Semantics

Synchronous Semantics: Environment interacts with the system after each step and time advances. This is conceptually quiet easy and appropriate for synchronous hardware. But, the system's reaction on the external input has to be simple (compare with semantics A).

Asynchronous Semantics: Synchrony Hypothesis: system may react with a chain reaction. External input only in **stable** states. Easier to model complex systems, abstraction from real-time. But, the implementation has to be shown to satisfy the assumptions of zero reaction time.