Chapter

10

Connections between the Functional and Structural Views

In Chap. 2 we discussed the functional view, described via the language of Activity-charts, and in Chap. 9 we discussed the structural view, described via Module-charts. The former depicts the system's decomposition into functional components, or activities, and the latter depicts its decomposition into structural components, or modules. This chapter discusses the connections between these two views and the way the connections are captured in our languages.

10.1 Relating the Functional and Structural Models

The functional view provides a decomposition of the system under development into its functional components, that is, its capabilities and processes. The structural view, on the other hand, provides a decomposition of the system into the actual subsystems that will be part of the final system, and which implement its functionality. The subsystems may be physical in nature, as were most of the modules in our description of the EWS example in Chap. 9, or logical in nature. For example, an `MMI` subsystem, which carries out all functions related to the man-machine interface of some system, would be considered a logical subsystem of that system.

We now describe the three types of connections between the functional and structural views: one is to describe the functionality of a module by an activity-chart (Sec. 10.1.1); the second is to allocate specific activities in an activity-chart to be implemented in a module (Sec. 10.1.2), and the third is to map activities in the functional description

of one module to activities in that of some other module (Sec. 10.1.3). The way these three kinds of connections are specified in our languages is described in Secs. 10.2, 10.3, and 10.4, respectively.

10.1.1 Functional description of a module

Our discussion of the functional view of the EWS in Chap. 2 centered around providing a functional description of the entire system, that is, the EWS module. However, there are a number of reasons for developing separate functional descriptions for some or all of the various submodules identified in the structural view:

- A module might represent an autonomous subsystem that is to be developed separately and then combined with the whole system (often with a relatively humble interface). For example, we may want to describe the SIGNAL_PROCESSOR of the EWS as a separate component. It may be used in other systems, and its independent description could be valuable for other purposes.
- A separate functional description of a submodule is sometimes a necessary prerequisite to its detailed design and implementation. Note that the description of the submodule's functionality may depend on a good understanding of the entire system specification, in which case a top-down approach is appropriate. For example, prior to the implementation of the CCU—the control and computation unit of the EWS—we might want to develop a separate description of its functionality. However, we can determine its specification only after identifying relevant functions in the entire EWS.
- It might be beneficial to obtain a good understanding of the functionality of a subsystem by identifying its capabilities to help carry out the functional specification of the entire system. In this case, a bottom-up approach is best. For example, we may prefer to first analyze the functionality of the MONITOR module, identifying the activities it will perform (such as GET_INPUT and DISPLAY_MESSAGE), and use these later, in the description of the processes that take place in the overall system. We shall discuss this approach further in Sec. 10.1.3.

In conclusion, we may wish to attach functional descriptions (i.e., activity-charts) to modules on different levels of the structural decomposition. See Fig. 10.1.

10.1.2 Allocating activities to modules

The structural decomposition and the identification of the flow of information between modules is part of the design stage of a system's development. But the design must be related to the system's functionality.

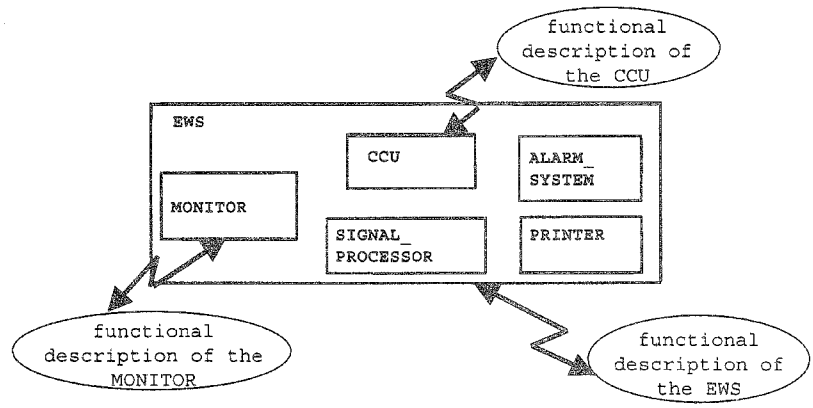


Figure 10.1 Functional descriptions attached to different modules.

That is, the functions identified in the functional view must be specified as being carried out by certain modules in the structural view. To capture this association, each of the functions must be allocated to one or more modules. In the EWS, for example, the SIGNAL_PROCESSOR performs the activity PROCESS_SIGNAL. This is a straightforward case of such an allocation. A more delicate case is the SET_UP activity, which contains subactivities that interact with the operator, as well as activities that carry out calculations. SET_UP should probably be divided among several modules with appropriate capabilities. Interaction would be carried out by the MONITOR, while the control of SET_UP and its calculation would be implemented by the CCU.

The allocation of activities to modules is the main activity carried out during top-level design. Indeed, some methodologies provide heuristic criteria for allocating activities to modules, for example by analyzing cohesion and coupling (Gomma 1993; Yourdon and Constantine 1979). This allocation actually determines the flow of information among the modules. Information that flows between two activities that are carried out by two modules will flow also between those modules. It is possible to examine alternatives for the allocation, using the amount of implied communication among the modules to decide which is best.

The allocation of activities to modules is also used in requirement traceability analysis. A functional requirement that was part of the original requirements of the system and was translated into an activity in the functional view will be automatically associated with the module that carries out that activity.

The allocation of activities to modules also allows restructuring of the functional description to define the implementation structure. One of the main criticisms of function-based decomposition methods such as Structured Analysis is that there is a troublesome discontinuity between the specification and design descriptions. This gap is

overcome to some extent in object-based methods, where both specification and design use the same components (objects) and the design is, in general, a refinement of the specification. This means that if the functional decomposition was carried out using an object-based approach, the mapping between activities and modules can be made easy: the decomposition into modules will use (or at least it will start with) the same components as the functional description.

10.1.3 Mapping activities to a module's activities

Sometimes it is not sufficient to allocate activities described on the system level to their implementing modules. We might want to be more concrete about the activities within the module's specification (as a subsystem) that are responsible for implementing the system activities. For example, the COMPARE activity is performed by the CCU, so there should be an activity within the CCU's functional description that implements the comparison. We could thus include an activity in the CCU's functional description, called CMP, which would be responsible for this. In such a case, we would map activities appearing on the system level to those appearing on the subsystem level.

This type of connection is even more useful in a bottom-up development process, where we first analyze the capabilities of each of the subsystems by developing their functional descriptions and later use them to construct the functional description of the entire system by detailing the scenarios in which these functions participate. Actually, the two views can be developed in parallel: while identifying the possible scenarios that occur during system operation, the required functions are defined and are specified as part of the appropriate subsystem. This approach is suggested by the ECSAM methodology, described in Lavi and Winokur (1989) and in Chap. 15 below. It is somewhat similar to an object-oriented analysis method in which the operations each object can perform are identified in parallel to the development of the scenarios (use cases) that use them. In Sec. 10.5 we illustrate this approach using the EWS.

In the following sections we show how our languages support the three connections discussed in the last three subsections.

10.2 Activity-Chart Describing a Module

The activity-chart EWS_ACTIVITIES, shown in Fig. 2.5, constitutes the functional description of the entire EWS system. In the structural view, the system appears as the top-level module of the module-chart EWS of Fig. 9.1. We may thus say that the *activity-chart* EWS_ACTIVITIES *describes* the functionality of the *module* EWS.

This connection between a module and its describing activity-chart is specified in the Data Dictionary entity of the module, in the field `Described by Activity-Chart` (see Fig. 10.2).

Notice that the connection is between an activity-chart and a module (and not between an activity-chart and a module-chart, or between an activity and a module). In our example, the module thus related is a top-level module, but this is not mandatory. It is possible to associate an activity-chart with any module in a module-chart. One reasonable way of proceeding (having already described the structural view of the system by a module-chart) would be to first describe the functionality of the entire system, that is, to construct a functional view for the top-level module, and then describe the detailed functionality of specific lower-level modules. Thus, in our example, we may now want to specify the activity-chart `CCU_AC` for the module `CCU`. The situation is illustrated in Fig. 10.3. (More about this issue in Secs. 10.4 and 10.5 and in Chap. 12.)

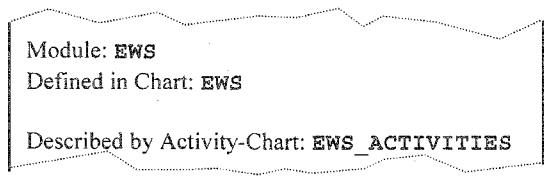


Figure 10.2 A module described by an activity-chart.

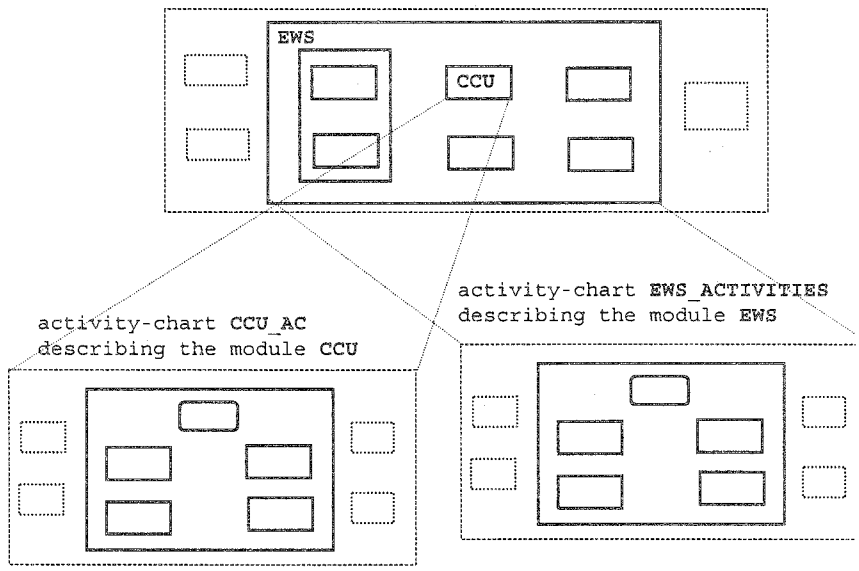


Figure 10.3 Activity-charts describing modules.

There must be a correspondence between the functional and structural decompositions of a module in terms of the environment and the interface with it. Because the top-level activity in the describing activity-chart represents the "functional image" of the module, we expect the external activities that interact with this top-level activity to correspond to the environment of the module described by the module-chart. When an external activity has been given a name, it must be the name of some module from the relevant environment. Indeed, as we saw in Fig. 2.5, the external activities in the chart EWS_ACTIVITIES were OPERATOR and SENSOR, the same as the modules external to the EWS module in the module-chart EWS. In this case, they are environment modules because EWS is the top-level module. However, in Fig. 10.4, the external activities in the activity-chart CCU_AC for the CCU module will be MONITOR, SIGNAL_PROCESSOR, ALARM_SYSTEM, and PRINTER, because they are the modules external to the module CCU, with which it interacts.

Notice that we included MONITOR as an external activity in CCU_AC but not its submodules KEYBOARD and SCREEN, although in the module-chart the CCU is connected to them through the communication lines. This is because the CCU is not supposed to "know" the internal structure of the modules with which it communicates.

Because the external activities in an activity-chart that describes a module correspond to modules, they have no entity of their own in the Data Dictionary, and they are viewed as pointers to the modules they

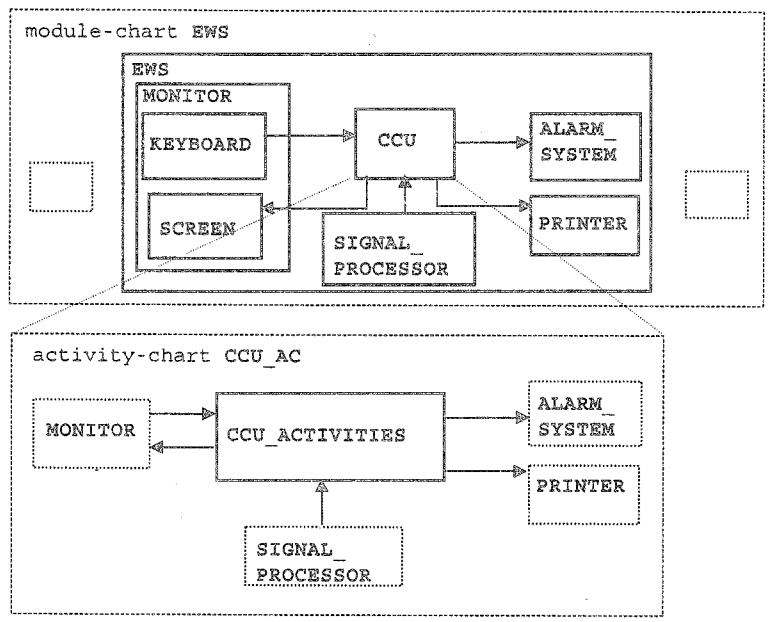


Figure 10.4 External activities corresponding to modules.

represent. Not only must the external elements of a module and its corresponding activity-chart match, but so must the information flowing in and out of them. To get a feeling for this requirement, compare Fig. 2.5 with Fig. 9.3. The former shows the information flowing to and from EWS_ACTIVITIES, and the latter shows the same for the EWS module in the module-chart. Most of the flows connect identically named external elements. However, notice that COMMANDS, RANGE_LIMITS, and SENSOR_CONNECTED were drawn in the activity-chart as flowing from OPERATOR, while in the module-chart they arrive from KEYBOARD (as components of USER_INPUT), and not from OPERATOR. This inconsistency arises from the fact that when we constructed the activity-chart we did not include the activity named GET_INPUT, for simplification. This activity is performed continuously in the MONITOR, whose role is to translate the KEY_PRESSING of the OPERATOR into COMMANDS and other information elements contained in USER_INPUT. To correct this problem, thus making the views consistent, we must add the GET_INPUT activity to the functional description. The revised version of the activity-chart EWS_ACTIVITIES of Fig. 2.5 that describes the module EWS is given in Fig. 10.5.

When constructing an activity-chart that describes a module, the names of the particular modules that produce or consume the externally flowing information may not be interesting. In such cases, the external activities can remain unnamed, as we illustrate in some of the following examples. However, as stated earlier, if an external activity is named, that name must correspond to a module in the corresponding module-chart.

10.3 Activities Implemented by Modules

Now that we are familiar with the general connection, whereby an activity-chart describes the functionality of a module in the module-chart, we can discuss how the components of each of these charts are related.

The relationship is this: all internal activities and control activities that appear in the activity-chart that describes a certain module are implemented by that module, and all the data-stores that appear in the chart reside in that module. In our EWS example, all activities in the EWS_ACTIVITIES chart (e.g., GET_INPUT, SET_UP, PROCESS_SIGNAL, etc.; see Fig. 10.5) are implemented by the EWS module, and the data-store LEGAL_RANGE resides in the EWS module.

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. This is done in the field Implemented by Module of the activity entity in the Data Dictionary, or in the field Resides in Module of the Data Dictionary

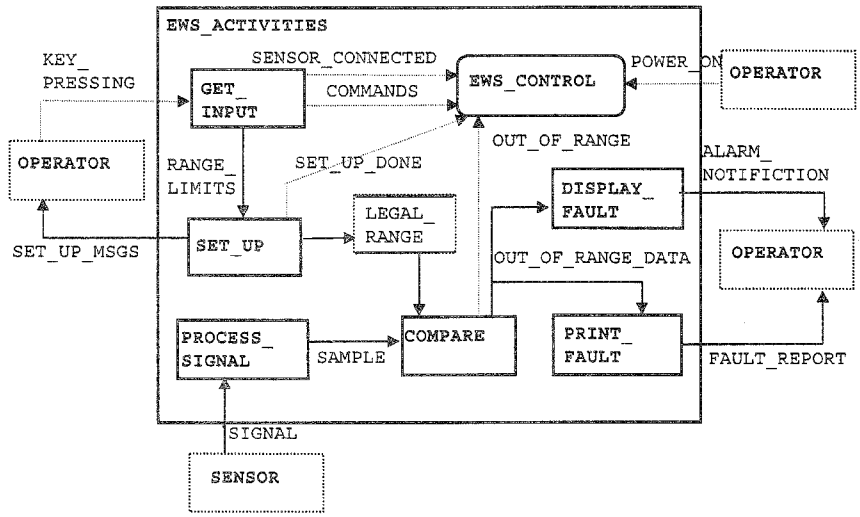


Figure 10.5 Revised activity-chart describing the EWS module.

entity for the data-store. For example, the PROCESS_SIGNAL activity is implemented by the module SIGNAL_PROCESSOR, and we have written this information in the Data Dictionary entity of the activity, as shown in Fig. 10.6. Similarly, the fact that LEGAL_RANGE resides in CCU appears in the Data Dictionary entity of the data-store.

Activities can be implemented by execution modules only (not storage or external modules), and data-stores can reside in any internal module, that is, in either execution or storage modules.

Several activities and data-stores can be allocated to a single module via the `implemented by module` or `resides in module` relation. For example, the activities COMPARE and EWS_CONTROL, as well as the data-store LEGAL_RANGE, are all allocated to the CCU module. However, a single activity or data-store cannot be distributed among several modules. In our example, the activities SET_UP, DISPLAY_FAULT, and PRINT_FAULT are each carried out by several modules. We could, of course, assign them to sufficiently high-level modules to cover this distribution, but this might lead to allocations that are too general to be useful. It is often better to further decompose such activities into subactivities that can each be allocated to a single module. This allocation will obviously be more informative. Thus, for example, SET_UP will be decomposed into PROMPT_RANGE, DISPLAY_SU_ERROR, VALIDATE_RANGE, and the control activity SET_UP_STATES. The role of the first two is to display messages, and they are implemented by the MONITOR module, while the other two are implemented by the CCU module. See Fig. 10.7.

The association of activities and data-stores with modules must be consistent with the module hierarchy and the activity hierarchy. As discussed earlier, all components of the top-level activity must be implemented in the module described by the activity-chart. Similarly, all subactivities and data-stores of an activity A that is implemented by a module M must be themselves implemented by M or its submodules. In other words, descendants of A cannot be allocated to modules outside of M. In the EWS example, we would not be allowed to specify that the SET_UP activity is implemented by the CCU and, at the same time, that its subactivity DISPLAY_SU_ERROR is implemented by the MONITOR module, because MONITOR is not contained in the CCU.

In Sec. 10.2 we discussed the consistency between the interface of the described module and the flows to the top-level activity. A similar consistency requirement applies to the flow of information on all levels. If two activities in the activity-chart are implemented by two different modules, we expect the information elements flowing between

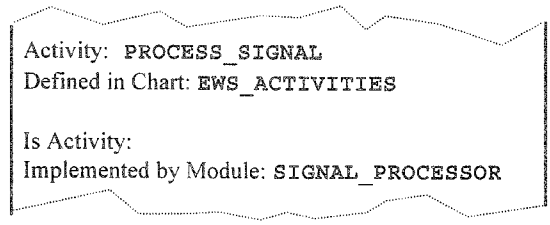


Figure 10.6 An activity implemented by a module.

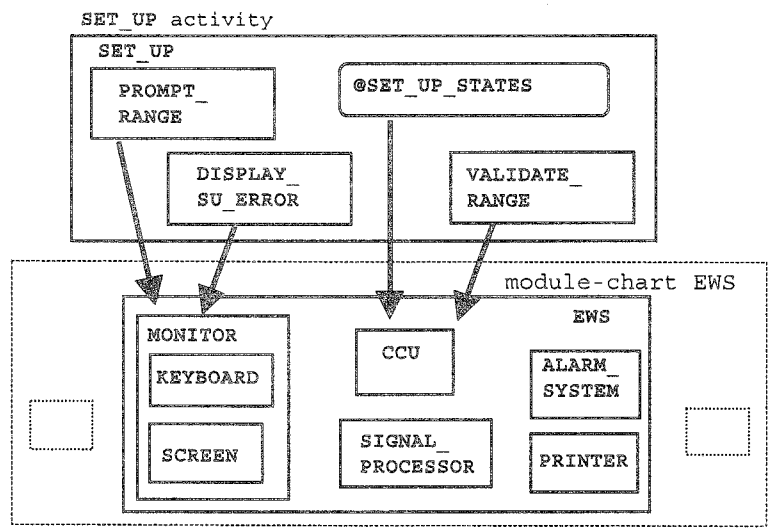


Figure 10.7 Allocation of subactivities of SET_UP to modules.

the activities to also flow between these modules. For example, compare Figs. 9.4 and 10.5. We allocated the PROCESS_SIGNAL activity to the SIGNAL_PROCESSOR module and the COMPARE activity to the CCU. In both charts the data-item SAMPLE flows between the corresponding boxes.

10.4 Activities Associated with a Module's Activities

This section deals with the possibility of mapping activities from the functional description of the entire system to activities from the functional description of its subsystems. The next example illustrates how this is actually done.

Figure 10.8 contains the activity-chart MONITOR_AC that describes the functionality of the module MONITOR. This module performs two functions, GET_INPUT and DISPLAY_MESSAGE, which are described, together with their inputs and outputs, in the activity-chart. (Some of the external activities are left unnamed in the figure because the sources and targets of the flowing data are not relevant here.)

Thus there are two activity-charts: EWS_ACTIVITIES for the entire system (EWS) and MONITOR_AC for one of the subsystems (MONITOR). In addition to allocating activities of the former chart to the EWS modules, we can also specify which activities in the latter chart correspond to these higher-level activities. In this example, we say in the Data Dictionary entity of the subactivity DISPLAY_SU_ERROR of SET_UP that it is activity DISPLAY_MESSAGE, implemented by module MONITOR. See Fig. 10.9. Similarly, the subactivity PROMPT_RANGE of SET_UP will also correspond to the activity DISPLAY_MESSAGE, using the field Is Activity. Attaching both activities to the same activity DISPLAY_MESSAGE means that the two will actually be implemented by the same function.

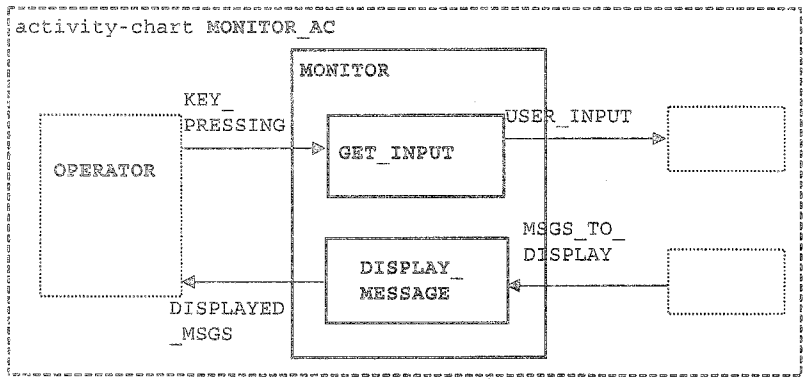


Figure 10.8 Activity-chart of MONITOR.

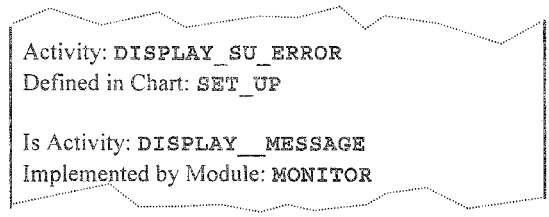


Figure 10.9 Mapping of activities by the is activity relation.

We also attach the activity GET_INPUT from the EWS_ACTIVITIES activity-chart to the activity GET_INPUT in MONITOR_AC. Although we use the same name for both activities, the field Is Activity must be specified. We say that PROMPT_RANGE in SET_UP is an *occurrence of the activity* DISPLAY_MESSAGE in the MONITOR module. The DISPLAY_MESSAGE activity is called the *principal activity* of PROMPT_RANGE.

Note that the field Is Activity is meaningful only when the Implemented by Module field is not empty. Moreover, the activity referred to must be one of the activities in the activity-chart that describes the implementing module.

In a similar way, a data-store may be associated with another data-store in the description of the submodules. The relevant field is Is Data-Store, which is completely analogous to Is Activity in the Data Dictionary entity for an activity. The terms used are the same: if a data-store P is defined as is data-store Q, then P is called an *occurrence of the data-store* Q, and Q is the *principal data-store* of P.

10.5 Object-Oriented Analysis with Module-Charts

Chapter 2 discussed an object-based approach to decomposition. This approach often fails to address one of the main goals of the specification phase because the decomposition alone makes it difficult to see the system's global behavior. Object-oriented approaches recommend that during requirement analysis, the behavioral scenarios (use cases) that might occur throughout the system should be identified, not just the objects and their operations. Here we show how the combination of module-charts and activity-charts and the Is Activity relation described earlier can be used to provide full specifications.

We shall use a module-chart to describe the system's objects. The operations of each object will be described as activities in the activity-chart that describes the module (object). The activity-chart that describes the top-level module (i.e., the entire system) will be used to describe the behavioral scenarios as sequences of object operations. An activity with its controlling statechart and subactivities will represent

a set of related scenarios, while the subactivities are mapped to the object operations by the `Is Activity` relation. Figure 10.10 illustrates this scheme.

The module-chart `EWS_OBJS` in Fig. 10.11 shows the decomposition of the `EWS` into objects and is similar to the one described in Sec. 2.1.3.

The operations of the `RANGE` object are described in the activity-chart `RANGE_OPS`, shown in Fig. 10.12. The Data Dictionary entry for the module `RANGE` contains the fact that it is `described by activity-chart RANGE_OPS`.

The activity-chart that describes the functionality of the entire system—the top-level module `EWS` in the figure—consists of the possible scenarios. The `SET_UP` scenario is the activity shown in Fig. 10.13; it consists of subactivities mapped to operations of the objects `RANGE` and `MMI_HANDLER`.

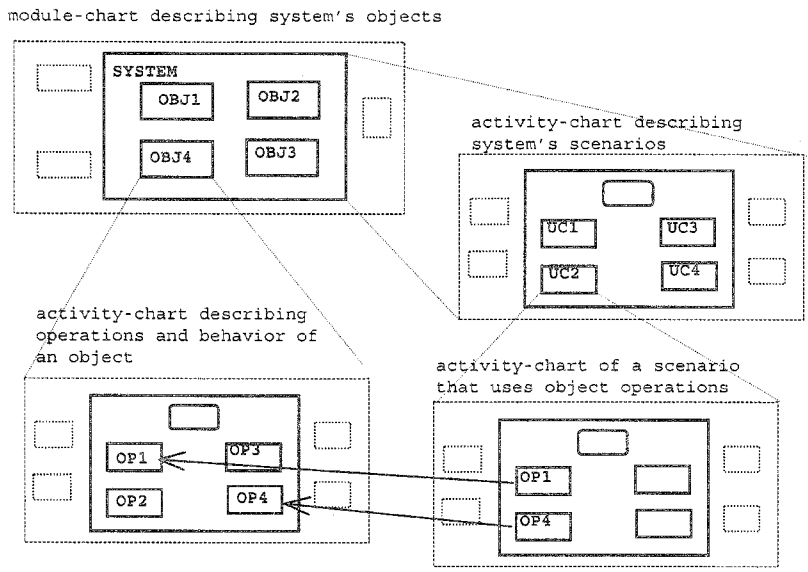


Figure 10.10 An object-oriented analysis model.

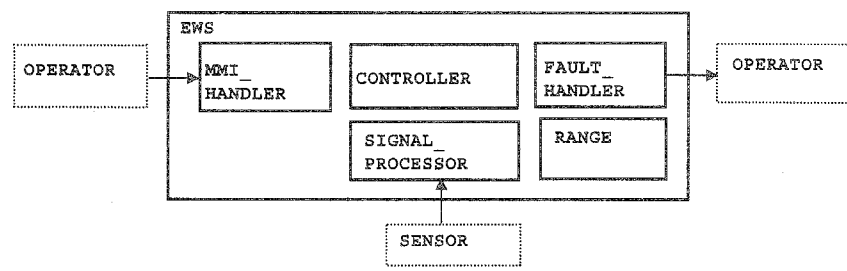


Figure 10.11 A module-chart based on object decomposition.

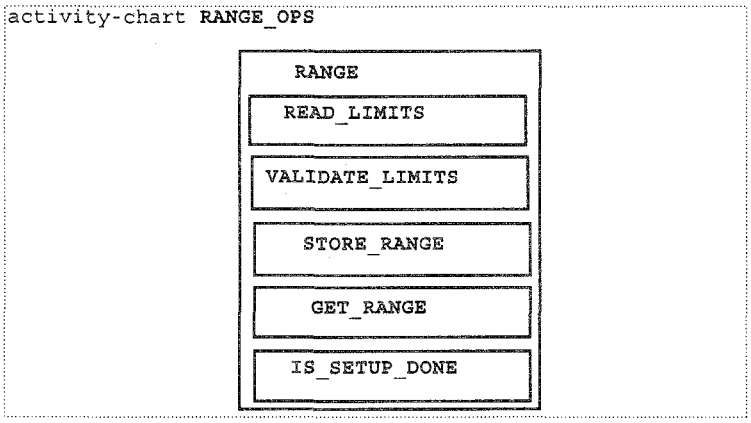


Figure 10.12 An activity-chart specifying the operations of RANGE.

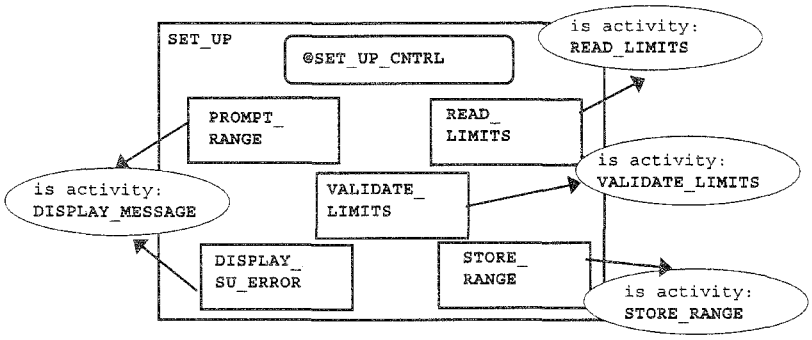


Figure 10.13 An activity-chart describing the SET_UP scenario.