Chapter

12

Putting Things Together

In the preceding chapters we discussed different kinds of charts and elements, and their interrelationships. A full-fledged model of a system may consist of many charts, each containing many elements. Now, although we have not yet described all the features of our languages, we pause here to take a bird's eye view and discuss how charts are connected to build a full model. Later, when we introduce additional features, such as generic charts, we will also address the issue of their location in the entire model. Do not be misled, however; when modeling a system, it is not necessary to specify all parts of the full structure as presented here.

This chapter also deals with entities external to the model—environment systems and testbenches. It discusses their role and how they relate to the other elements of the model.

Charts that make up the model share elements among themselves. Therefore, the picture is incomplete without the material of Chap. 13, in which we discuss the scope of elements and their visibility with regard to the various components of the model. We also introduce there another component of a model, the *global definition set,* which contains information that is visible to the entire model.

12.1 Relationships among the Three Kinds of Charts

We now describe the full picture of our EWS example, as it emerges from the various pieces described in earlier chapters. The fact that our exposition follows a certain order is not meant to imply any specific order recommended in developing the model.

The interface of the EWS with its environment and its structural decomposition appear in the module-chart `EWS` of Fig. 1.7, which is also

shown on the left-hand side of Fig. 12.1. The entire system is depicted by the top-level module therein, named EWS. The activity-chart EWS_ACTIVITIES, whose contents is shown in Fig. 1.4, describes the functionality of this top-level module. The top-level activity in that chart, EWS_ACTIVITIES, corresponds to the EWS module, so the interfaces of the two must be the same. See Chap. 10.

Control activities appearing in an activity-chart are described by statecharts. See Chaps. 6, 7, and 8. Thus in Fig. 12.1 we see that the control activity of the activity EWS_ACTIVITIES is described by the statechart EWS_CONTROL of Fig. 1.6. Similarly, the control activity of SET_UP is described by the statechart of Fig. 7.3.

We refer the reader to App. B, which contains the entire EWS model.

As we saw in Chap. 10, an activity-chart can be attached to any module in the module-chart as its functional description. The control activities in these activity-charts are also described by statecharts. For our EWS example, this results in the structure shown in Fig. 12.2.

Figure 12.2 captures only the relationships between the three types of charts that describe the three views. However, as explained in Chap. 11, each of these logical charts can be decomposed into several physical charts, thus creating a more complex network of charts. These additional connections are based on the three types of relationships described therein: one, a module described by an activity-chart, is specified in the module entry in the Data Dictionary, and the other two, that between a control activity and its describing statechart, and

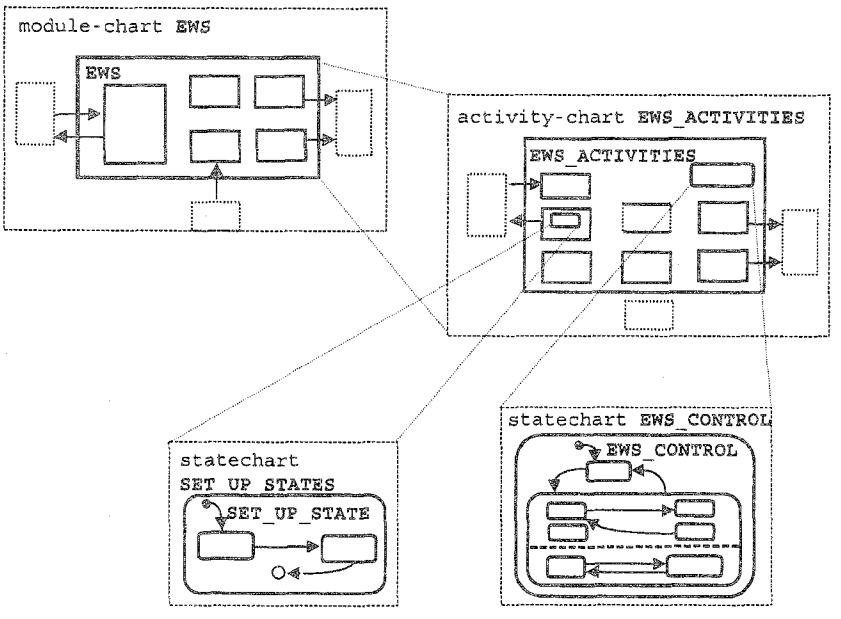


Figure 12.1 The charts of three views of the EWS.

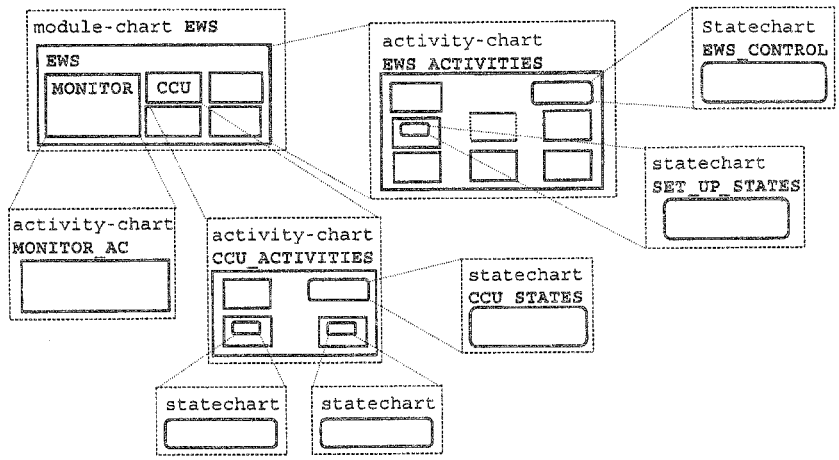


Figure 12.2 Charts in multilevel specification of the EWS.

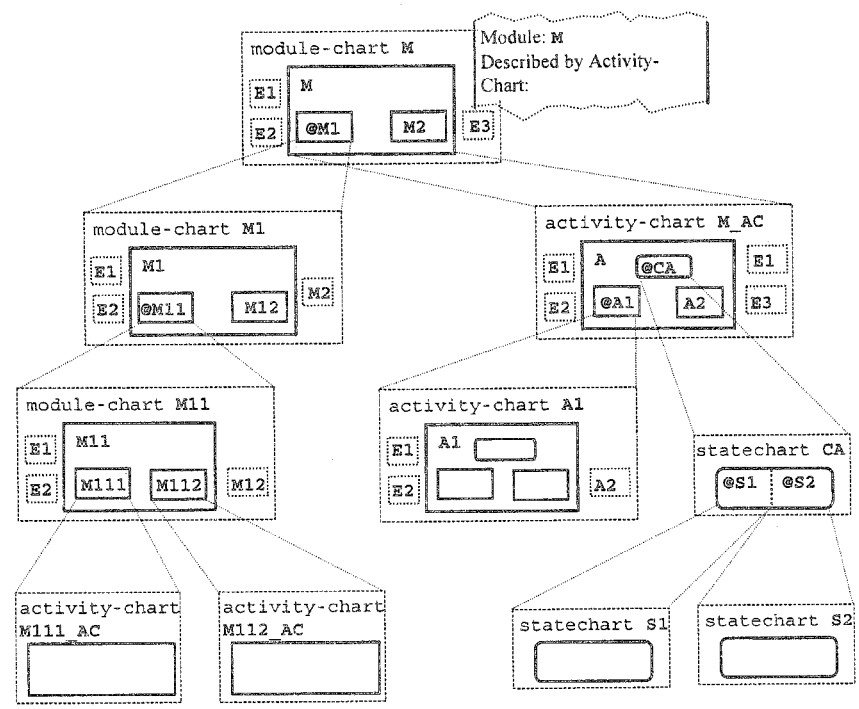


Figure 12.3 Relations between charts in system specification.

the offpage (decluttering) relationship, are depicted graphically, using the @ symbol.

A schematic example of a structure built up from many of these relationships is shown in Fig. 12.3. Notice that this particular figure contains only one logical module-chart, consisting of the three physical

charts M, M1, and M11, but three logical activity-charts, namely, M_AC, M111_AC, and M112_AC.

12.2 A Chart in a Model

Regarding the terms *logical chart* and *physical chart,* from here on, we mostly use *chart* to mean physical chart. Each (physical) chart plays a role in the whole specification according to its relationships with other charts. The top-level box of the chart is its subject. For example, the activity-chart EWS_ACTIVITIES of Fig. 10.5 describes the functionality of the EWS module. Its top-level activity is EWS_ACTIVITIES, which is therefore its subject. In our examples we almost always use the same name for the chart and its top-level activity, although this is not mandatory.

Charts will always be identified by name. Chart names must be unique throughout the entire model, even those of different types. Thus we may not have a module-chart and an activity-chart with the same name in a single model.

Like other elements in the model, a chart has an associated entry in the Data Dictionary. This entry contains descriptive information, such as short and long descriptions and attributes. It may also contain administrative information, such as the owner of the chart and its creation date, version number, and access privileges. We shall see later that this entry is also used to define a chart as generic, that is, as one that can be instantiated multiple times in the model.

12.3 Hierarchy of Charts

The relations between boxes and charts induce a *hierarchy of charts.* A chart is considered to be a *parent chart* of all the charts that describe its boxes by the offpage chart relation, by the relation between a control activity and its statechart, and by the module described by activity-chart relation. Referring to Fig. 12.3, for example, we find that the module-chart M is the *root* of the hierarchy; it is the parent of the module-chart M1 and the activity-chart M_AC. The activity-chart M_AC, in turn, is the parent of the activity-chart A1 and the statechart CA, and the statechart CA is the parent of statecharts S1 and S2. As in other cases, here, too, we use the terminology *subchart, ancestor,* and *descendant.* Thus, for example, the module-chart M1 is a subchart of M, and all the charts in Fig. 12.3, except for M itself, are descendants of M.

The chart hierarchy is sometimes called the *static structure of charts.* The structure for the example of Fig. 12.3 is shown in tree form in Fig. 12.4. The chart hierarchy serves as a sort of table of contents for the specification.

The uniqueness rules discussed in earlier chapters (e.g., that each chart can be a definition chart of a single box only) imply that each chart

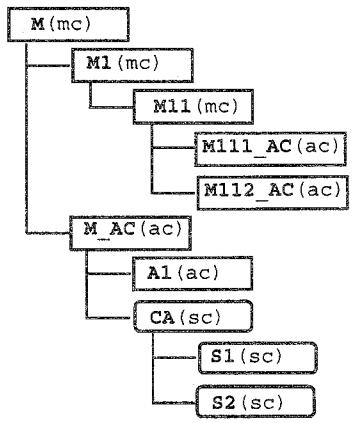


Figure 12.4 Hierarchy of charts.

has (at most) one parent. In addition, cyclic definitions are not allowed, so that the hierarchy of charts will indeed be either a tree (as in Fig. 12.4) or a forest of trees. Now, in a typical full specification there is usually a module-chart that describes the system context and sometimes the top levels of the structural decomposition, too, and all the other charts are its descendants. This renders that module-chart the root chart, so that the chart hierarchy is a single tree. However, in many cases, especially if the specification is carried out in a bottom-up manner and is not yet complete or when using methodologies that do not call for a single module-chart for the context description, there might be no such root, and the structure will therefore be a forest. Moreover, we shall see later that generic charts, those that can be instantiated multiple times in the model, have no parents and are considered roots in the chart hierarchy, so that here, too, the structure will be a forest. A tree in the chart hierarchy is sometimes called a *cluster*; in Fig. 12.4, the entire structure consists of a single cluster.

12.4 Entities External to the System under Description

The model that specifies the system under development operates in the context of the environment systems. We now discuss these systems and other external entities that are connected to the system model and might interact with it.

12.4.1 Environment modules or activities

A number of times we stated that the external boxes in a chart represent either boxes in the parent chart or parts of the real environment of the model. The EWS example, as presented throughout this book,

models the context of the system by the top-level module-chart EWS. This is the root of the chart hierarchy, and, as always with the context module-chart, all of its external boxes (in our case, OPERATOR and SENSOR) are *environment modules* and are not part of the system. In a typical model, all other module-charts are offpage charts, whose external modules are occurrences of modules from their parent chart. For example, if the MONITOR's structure is specified in a separate off-page module-chart, this chart will contain two external modules, the CCU and the OPERATOR, which are simply occurrences of the two modules that appear in its parent chart, the module-chart EWS. See Fig. 12.5.

In Fig. 12.5, we also show the activity-chart EWS_ACTIVITIES, which describes the top-level module EWS (see Sec. 10.2), and which, as such, is a subchart of the EWS module-chart. Its external activities OPERATOR and SENSOR are simply occurrences of the corresponding environment modules from the parent module-chart. Other offpage activity-charts participating in the functional description, such as the SET_UP chart in Fig. 11.2, also contain external boxes that are linked to other activities and data-stores from the parent chart (e.g., GET_INPUT, LEGAL_RANGE, and OPERATOR). However, a model does not necessarily contain a module-chart. One can construct the functional view only, starting with a root activity-chart that will contain the environment systems, too.

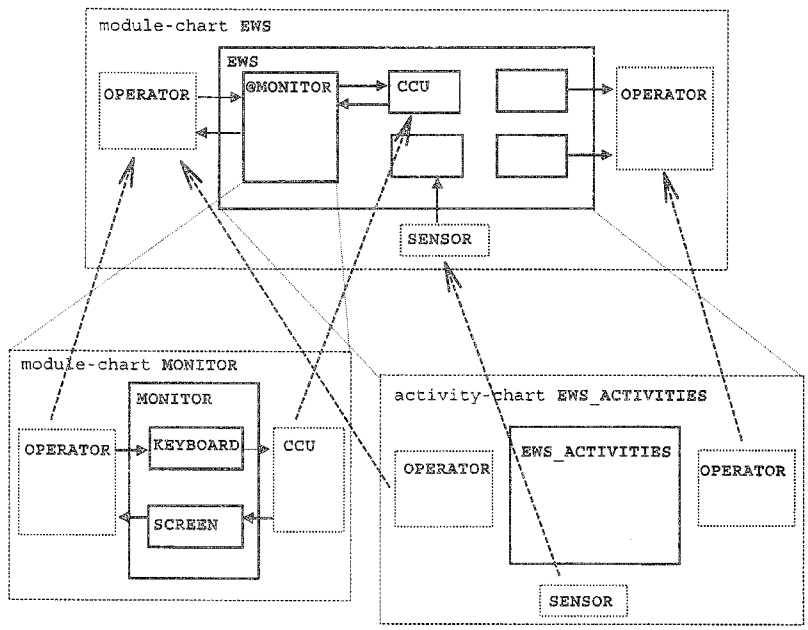


Figure 12.5 External and environment boxes.

An environment box—module or activity—has an entry in the Data Dictionary, but an external box that points to another box has no entry of its own. The Data Dictionary entry of an environment box may contain descriptions and attributes, but not behavioral information. For instance, a mini-spec cannot be associated with an environment activity. In fact, when modeled as external entities, the environment systems cannot be associated with functional and behavioral descriptions in our languages at all. It is impossible to associate an activity-chart or a statechart with an environment module or activity. Often there is only limited and imprecise knowledge about the external entities. However, in some cases there are assumptions about the behavior of the interface signals that are significant to the design of the system, and the designer might want to express them explicitly. This can be done by including the relevant environment systems as part of the model and representing them as internal modules or activities. It helps to give them some user-defined marks to indicate that they are beyond the scope of the system under development. This technique can also be used when the designer wants to simulate the system in its environment and wants to use the modeling languages to describe the external systems. It is often convenient to specify environment behavior in a statistical manner, for which purpose one can use the random functions listed in App. A.3.

The ECSAM methodology, which essentially employs our modeling languages (Lavi and Winokur 1989), has been extended recently to construct what its authors call *a black box external model* by including the environment systems in the model, as we suggest here (Lavi and Kudish 1996).

Sometimes it is easier to use a conventional programming language to simulate the external systems, particularly when these systems have already been implemented in software. In general, any existing implementation can be used for simulation and prototyping purposes. The value of supporting tools based on our languages can be enhanced if they can be made to provide means for linking the model execution facilities to an external existing environment.[1]

12.4.2 Testbenches

Other external entities that interact with a typical model are the tests developed to check its behavior. These tests are valuable even beyond their primary purpose, which is to check whether the model matches some preliminary requirements and behaves as expected. Sometimes the model is built as a reference model, that is, it is to be compared with its implementation. In such a case, the model is developed for

[1] STATEMATE indeed provides such means.

prototyping purposes, and the real system is developed later, independently, with the intention that it behave similarly. Hence, tests that are developed to check the model can be used later to check the implementation. Extensive testing of the model is even more justified when it is automatically transformed to yield an implementation. In this case, if the model fulfills the requirements and is found to be correct by the tests, then the synthesized implementation is correct, too.

One approach to testing the model is based on generating test scenarios according to some patterns and rules, by a special-purpose test driver (written as an external program or with the aid of our modeling languages). The outputs of the modeled system are then collected by some monitoring function, and the collected data can be analyzed and checked in order to learn about the system's behavior and performance and to detect undesired reactions.

Another approach uses auxiliary charts (mainly statecharts) to express and verify temporal requirements that are related to the model, such as safety and liveness properties (Manna and Pnueli 1992). These special charts are called *testbenches,* or sometimes *watchdogs,* and we now illustrate how they are used.

Assume that we want to be convinced that the EWS model satisfies the *causality property* that an alarm is issued only after an out-of-range situation has been detected. This requirement is expressed in terms of our model as follows: the activity DISPLAY_FAULT operates (is started) only after the event OUT_OF_RANGE has occurred. We can now construct a testbench statechart, ALARM_CAUSALITY, shown in Fig. 12.6, that will run in parallel with the system model and will "watch" the model execution under different scenarios of external changes. Whenever the requirement is violated by some scenario, the testbench will enter the state ERROR.

This testbench checks for the kind of causality categorized as a *safety property* in the literature on program verification. Safety properties often take the form *B never occurs after A* (Manna and Pnueli 1992). In such a case we look for a scenario that violates the property (i.e., one

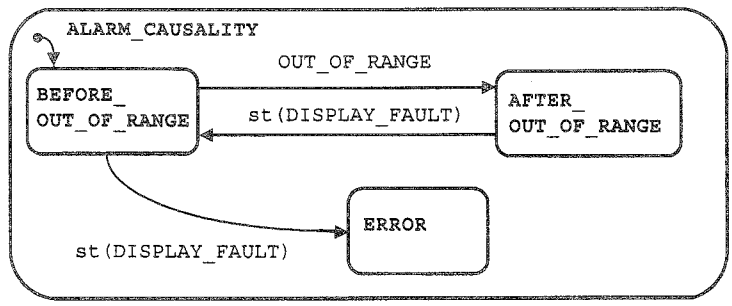


Figure 12.6 The testbench statechart ALARM_CAUSALITY.

in which B occurs after A) to prove that the model does *not* satisfy the requirement. A similar technique can be used to check whether the model satisfies what is called a *liveness property.* One variation of liveness states that *after A occurs, B can occur*. To convince ourselves that this requirement is satisfied, we draw a testbench in which a scenario of B after A leads to a success state.

A supporting tool (such as STATEMATE) can be instructed to try out many scenarios, perhaps even all of them exhaustively, to find one that satisfies or violates such requirements.

Testbench statecharts are not an integral part of the model and the hierarchy of its charts. Due to their special role, they are allowed to refer to the model's elements without necessarily obeying the scoping rules discussed in Chap. 13. For example, in Fig. 12.6, the testbench chart ALARM_CAUSALITY refers to the activity DISPLAY_FAULT, although this violates the visibility rules defined in Chap. 7 for activities.

In terms of the scoping rules, the difference between environment modeling and using testbenches is analogous to two different ways of testing a hardware board: the former has a well-defined interface and is therefore like connecting to a board via the connector's pins, and the latter is less disciplined and therefore more like monitoring a signal with a probe.