Chapter

14

Generic Charts

Chapter 11 discussed the possibility of describing the contents of a box element in a separate offpage chart. An offpage chart can describe only a single box. In this chapter we introduce generic charts, which are reusable components that may have multiple instances in a model. In other words, a generic chart can be used to describe the contents of several similar boxes.

Generic charts are linked to the rest of the model via parameters; no other elements (besides the definitions in global definition sets) are recognized by both generic charts and other portions of the model.

In this chapter we describe how generic charts and their formal parameters are defined, how they are instantiated in the model, and how the actual elements are bound to their formal parameters.

14.1 Reusability of Specification Components

Many kinds of systems give rise to cases requiring a number of similar components. For example, assume that the EWS monitors several sensors, each with its own processing function, and all with the same pattern, as shown in Fig. 7.12. The new activity-chart with the multiple sensors is shown in Fig. 14.1*a*. The function `PROCESS_SIGNALS` contains five similar activities, `PS1` through `PS5`, which process `SIGNAL_1` through `SIGNAL_5`, respectively. Each `PSi` is described by a separate Data Dictionary entry, similar to the one shown for `PS1` in Fig. 14.1*b*. The output of the function dealing with the *i*th sensor is sent to the `COMPARE` function via the *i*th component of the array `SAMPLES`. The functions vary in the sampling interval, `SAMPLE_INTERVAL_i`, and in the constant factor `K_i` that multiplies the sampled signal.

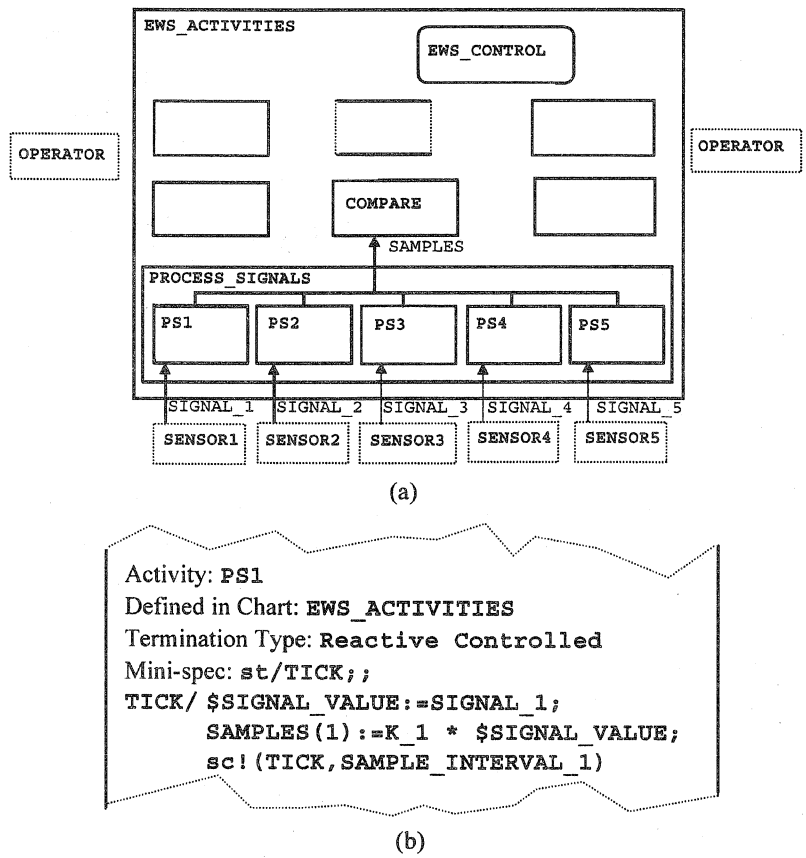


Figure 14.1 Processing multiple sensors in the EWS.

It is quite obvious that this solution is not efficient. In addition, it does not make it clear to a viewer that the components are essentially identical. There should be a way to specify a repetition of the same component many times, as in electronic and software design, defining the detailed contents only once and using the component generically wherever needed. For this purpose our languages provide the mechanism of *generic charts.*

We saw that when used the various components can differ in the details of their connections with the outside world (i.e., in the data elements through which they exchange information), as well as in the internal settings that determine the nature of each specific instance. Both will be handled by *parameters.*

The generic chart mechanism can be used to model electronic designs with repeating components, and software systems that contain multiple objects of the same class.

14.2 Definition and Instances of Generic Charts

In a sense, generic charts are similar to offpage charts: in both cases we draw an empty box and point to another chart that describes its contents. Here, however, we can specify repetition; indeed, sometimes we draw an offpage chart first, later realize that we really want to repeat the specified portion, and switch to a generic chart. The similarities and differences between these two mechanisms are discussed next.

14.2.1 Notation and basic rules of generics

A chart can be defined as a generic chart in its Data Dictionary entry, and it can then have multiple *instances* in the model. An instance of a generic chart is sometimes called a *generic instance,* to distinguish it from an *offpage instance.* To apply this to the EWS example, we define an activity-chart `PROCESS_SIGNAL` and specify it as generic. Its top-level activity `PROCESS_SIGNAL` has a Data Dictionary entry. This is shown in Fig. 14.2*a*. We then specify instances of this generic activity-chart (`PS1` through `PS5`) inside `PROCESS_SIGNALS`, using the symbol `<`, as in `PS1<PROCESS_SIGNAL`. See Fig. 14.2*b*.

While this example was of a generic activity-chart, we can also have generic module-charts and generic statecharts. The former can be useful when the system is built of similar modules, such as multiple signal processors, and the latter are often used to describe similar orthogonal components, as we shall see shortly.

Because generic charts usually appear in different contexts, the external boxes in a generic chart are not allowed to point to any particular boxes in the model and are very often left unnamed. In Fig. 14.2*a*, the `SIGNAL` comes from some "generic" `SENSOR`, while the activity's output flows to an unknown target, which is why the external box is left unnamed.

Regarding names, the box name can really be omitted in an instance (and we can thus write, say, `<GEN`), as is usually done for offpage instances. However, here this is not recommended, and it is only possible when the instance is unique on that level of decomposition. More commonly, instance activities have their own individual names, as in `PS1`, `PS2`, etc. The name of the generic chart should not appear when we refer to the instance, so we write expressions such as the start action `st!(PS1)`.

The instance box must be basic, that is, it may not contain subboxes. Moreover, it cannot contain any behavioral information, and this applies to all three kinds of charts: instance states in a statechart cannot have static reactions and attached activities, instance activities in an activity-chart cannot have mini-specs and combinational assignments, and instance modules in a module-chart cannot be described by

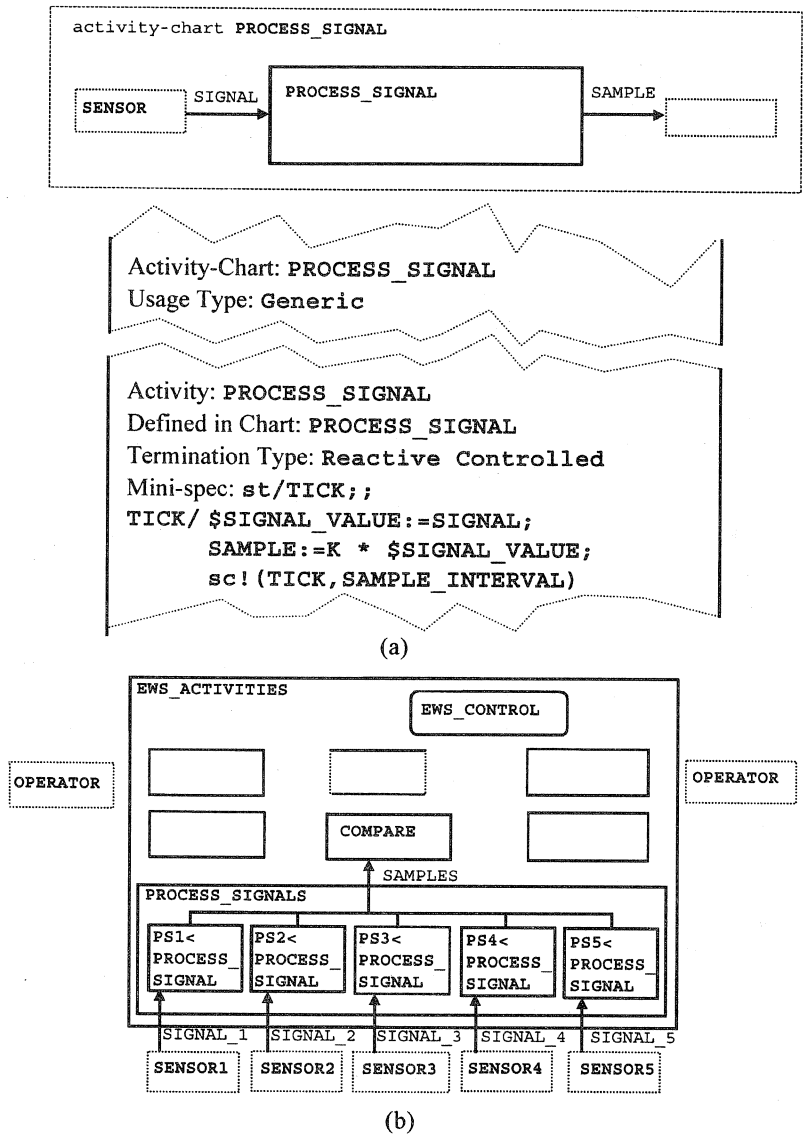


Figure 14.2 (*a*) A generic activity-chart and (*b*) its instances.

an activity-chart. All such information is inherited by the instances from their describing generic charts.

In modeling a generic chart, one may include instances of offpage charts, as illustrated in Fig. 14.3. Note that this reference to the offpage chart COMPUTE in the generic chart PROCESS_SIGNAL implies that there will be multiple occurrences of COMPUTE in the full expansion of the model, but each of them will belong to a different scope.

A generic chart may also contain instances of other generic charts, but care must be taken to avoid cyclic instantiation thereof.

The notion of resolution is applied to charts just as it is applied to other elements in the model; that is, a reference to a chart appearing in an instance name will be resolved to a chart of appropriate type whose definition in the Data Dictionary matches the reference. Therefore, if an ordinary activity, for example, is named `A>GEN`, `GEN` must be defined as a generic activity-chart. Similarly, a chart defined as generic cannot be used as an offpage chart or as an activity-chart describing a module. This means that if `GEN` is a generic chart, no box can be named `A@GEN`, and `GEN` cannot appear in the Data Dictionary entry of any module in the field `Described by Activity-Chart`. A model containing instances of charts that do not yet exist is incomplete, and in such a case the same chart cannot appear both as a generic and an offpage instance. For example, `A1>A` and `A2@A` is an inconsistent situation, even when `A` is not yet defined; any completion of such a model would be illegal.

14.2.2 Generic charts in the chart hierarchy

In Chap. 12 we discussed the hierarchy of charts. We saw that this hierarchy is based on several kinds of relationships between a box and a chart: an instance of an offpage chart, a control activity described by a statechart, and a module described by an activity-chart. The hierarchy of charts in Fig. 14.4 is derived from the components of the EWS model appearing in Fig. 12.2.

The hierarchy of charts defines the visibility scope of textual elements and has a dominant role in the resolution algorithm. Generic charts have no parent charts; each generic chart is the root of a tree that it induces in the chart hierarchy. The tree itself is defined as in the ordinary case. Therefore, according to the visibility rules and the resolution algorithm (see Chap. 13), generic charts do not recognize elements from other clusters. Instead, they share elements with the rest of the model via parameters, as we shall see. In addition, like all portions of the model, generic charts can see the data in the global definition sets, and they may thus use the definitions of constants and user-defined types appearing there.

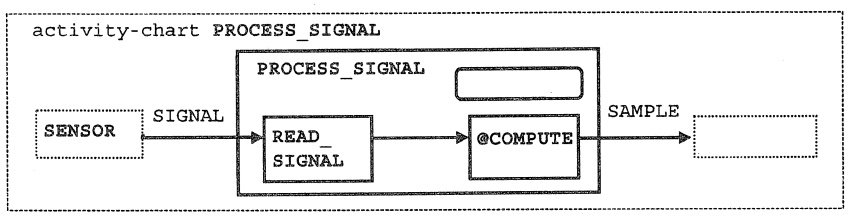


Figure 14.3 A generic chart containing an offpage chart.

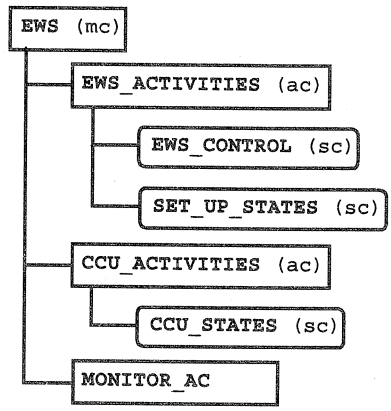


Figure 14.4 The hierarchy of charts for the EWS model.

The external boxes in a generic chart cannot be resolved to other boxes in the model because there is no parent chart to which they can refer. As mentioned earlier, they are usually left unnamed.

Because the chart hierarchy is determined by making an offpage chart an offspring of the chart in which it is referred to, this structure is actually a kind of "table of contents" for the model that shows where charts are used. When generic charts are involved, the chart hierarchy is expanded to a *chart usage hierarchy*[1]; the generic charts appear under the chart in which they are used (instantiated), just as offpage charts do, but the special symbol < is used to distinguish them from the others and to emphasize the fact that they do not participate in the resolution algorithm. Note that a generic chart can appear along several branches of the hierarchical structure as a leaf. Because a generic chart may have many instances in the same chart, it can be useful to provide the number of instances near the chart name.

As an example, let us assume that the chart `EWS_ACTIVITIES` contains five instances of the generic chart `PROCESS_SIGNAL`, as described earlier, and that this generic chart contains an instance of the offpage activity-chart `COMPUTE`. Figure 14.5 contains the tree of Fig. 14.4 enhanced with these additional components. Note that this usage hierarchy contains two separate trees.

14.3 Parameters of Generic Charts

Parameters are used to characterize the particular instances of a generic chart and to link it to its environment. Parameters are the

[1]In STATEMATE the chart usage hierarchy is called the *model tree.*

main means by which an instance of a generic chart is able to share data with the rest of the model.

14.3.1 Formal parameters of a generic chart

Each generic chart has a set of *formal parameters*. These are either *ports* (i.e., channels, through which information flows in and out of the component) or *constant parameters* (i.e., values used to characterize the particular instance at hand). The parameters are defined explicitly by the specifier in the Data Dictionary entry of the generic chart. They are given by their name, element type (event, condition, data-item, or activity), and mode (`constant` or one of the three port modes: `in`, `out`, and `in/out`). Each formal parameter has a Data Dictionary entry in which more information about the element can be added, such as its structure and data-type.

The generic chart `PROCESS_SIGNAL` described earlier has an `in` port `SIGNAL`, and an `out` port `SAMPLE`. In addition there are two `constant` parameters, `SAMPLE_INTERVAL` and `K`, that make it possible to set some values differently for each individual instance; the first influences the sampling rate of the sensor, and the second is used to calibrate the sampled value. The Data Dictionary entries of the generic chart, including its parameters, are shown in Fig. 14.6.

Port parameters can be of any data-type and structure. Array parameters can be defined with or without an index range. The index range

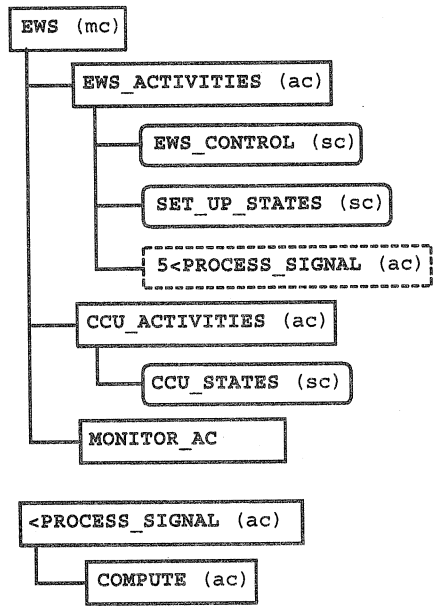


Figure 14.5 The chart usage hierarchy of the enhanced EWS.

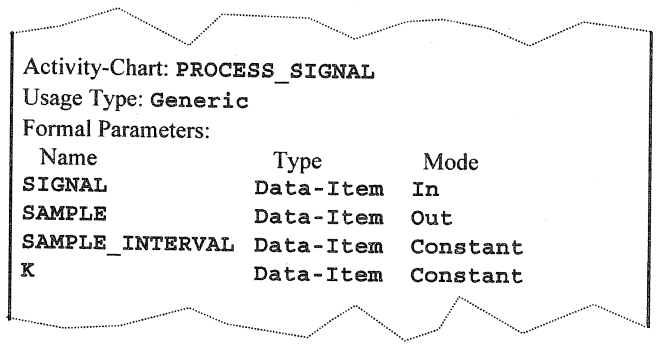


Figure 14.6 Definition of formal parameters of a generic chart.

definition can use literal or named constants, constant parameters, or it can be based on an index range (e.g., parameter `A` is `array 1 to length_of(A) of integer`, which defines an array parameter whose upper range index is equal to the length of the actual binding). When the array parameter is defined without an index range, the index limits are inherited from the actual binding, a notion defined in the next section.

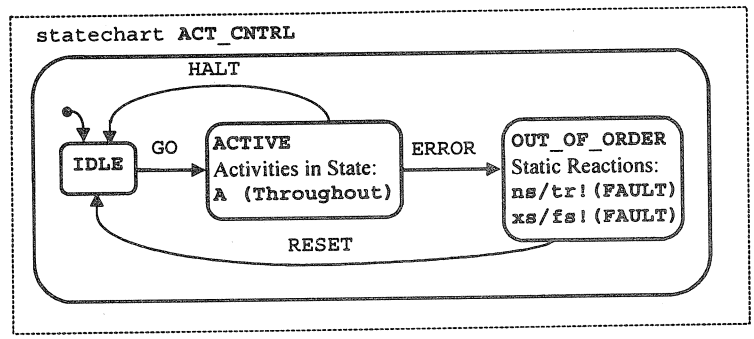
A generic chart that communicates with its environment via a queue will have a queue `in/out` parameter. The example shown in Fig. 8.8, in which four EWS instances communicate with two printers, can be modified so that it uses generic charts for the EWS and the printer, and each of these has a queue parameter.

Record and union parameters must be defined with a user-defined type that has the desired structure. This follows from the rule that the actual bindings must be consistent in type with the formal parameter, and two records/unions are consistent only if they have the same user-defined type. This issue is discussed further shortly.

The formal parameters are used inside the generic cluster like any other element, but their usage must be consistent with the parameter's mode: the value of an `in` parameter is expected to be used by the component, while that of an `out` parameter should be affected by the component. The value of an `in` parameter may be modified, as long as the modified value is not used later on outside the component.

Constant parameters can be used in places where constants are allowed: constant parameters can appear, for example, in the definition of an array index range, but they cannot label a flow-line or be assigned a value in an assignment action. Very often, the instances of a generic chart are arranged in an array, and an integer constant parameter is used to identify the individual instances. For example, in models of client/server architectures when multiple similar clients send messages to a server via a queue, each client can be an instance of a generic chart that identifies itself by its index.

Statecharts (but not activity-charts or module-charts) may have parameters of type activity. An activity parameter is considered to be an `in/out` port, the idea being that the component can send a control signal to an activity (e.g., `st!(A)` or `sp!(A)`), and can sense its status (e.g., by `sp(A)` or `ac(A)`). For example, assume that in the processing unit of a mission-critical system there are several components that perform a similar function. (Such redundancy is often incorporated to enhance reliability.) Each of these components has the same behavioral pattern, which is specified by the generic statechart `ACT_CNTRL` of Fig. 14.7. The formal parameters include the input events that trigger transitions, and the activity `A`, which is activated by the statechart. This pattern can control any activity that is bound to the formal parameter when the chart is later instantiated. The input events consist of control commands (`GO`, `HALT`, and `RESET`) and an indication of an error in the input device. The `out` parameter `FAULT` is used to report the status of the particular instance. A sample usage of this generic statechart is illustrated in Fig. 14.7.



Statechart: **ACT_CNTRL**
Usage Type: **Generic**
Formal Parameters:

Name	Type	Mode
GO	Event	In
HALT	Event	In
RESET	Event	In
ERROR	Event	In
A	Activity	In/Out
FAULT	Condition	Out

Figure 14.7 A generic statechart with an activity parameter.

14.3.2 Actual bindings of parameters

For each instance of a generic chart there is a binding of actual elements to the formal parameters. Ports can be bound to variables or aliases; in fact, any data element that labels a flow-line (with the exception of an information-flow) can be bound. The port binding is analogous to connecting components ports with signal lines in an electronic scheme. Every change in the actual element will be available immediately to the instance, and every change inside the instance will be sensed outside by the connected elements. Constant parameters are bound to constant values, that is, literal constants, named constants, or operators that yield constant values, such as those that relate to the index range of arrays (e.g., `length_of(A)`).

The binding information is supplied in the Data Dictionary entry of the instance. Figure 14.8 shows the parameter bindings for the instances `PS1` and `PS2` of the generic chart `PROCESS_SIGNAL`, whose formal parameters were defined in Fig. 14.6. The different bindings to the constant parameter `SAMPLE_INTERVAL` determine different sampling rates in each component. `HIGH_RATE` and `LOW_RATE` are two named constants; they can be defined, for example, in a global definition set. The `in` and `out` ports are bound to the actual data-items—the signal that comes from the sensor and the corresponding component in the array of `SAMPLES`.

As another example, we instantiate the generic statechart `ACT_CNTRL` of Fig. 14.7 three times in the statechart `PROC_CNTRL`. The purpose of the containing statechart is to activate three copies of an activity, each

Activity: **PS1<PROCESS_SIGNAL**
Defined in Chart: **EWS_ACTIVITIES**
Actual Bindings:

Name	Type	Binding
SIGNAL	Data-Item	SIGNAL_1
SAMPLE	Data-Item	SAMPLES(1)
SAMPLE_INTERVAL	Data-Item	HIGH_RATE
K	Data-Item	0.5

Activity: **PS2<PROCESS_SIGNAL**
Defined in Chart: **EWS_ACTIVITIES**
Actual Bindings:

Name	Type	Binding
SIGNAL	Data-Item	SIGNAL_2
SAMPLE	Data-Item	SAMPLES(2)
SAMPLE_INTERVAL	Data-Item	LOW_RATE
K	Data-Item	1.0

Figure 14.8 Activity instances and actual parameter bindings.

processing a signal from a different sensor. The statechart also continuously monitors the status of these activities, and when all of them fail it issues a fault alarm. See Fig. 14.9, which shows the activity-chart that contains `PROC_CNTRL`, the statechart itself, and the Data Dictionary entries of the state instances. These entries contain the actual parameter bindings; in particular, they contain the binding to the activity parameter.

The actual binding must have the same type and structure as the formal parameter. In particular, in the case of data-items the following rules hold:

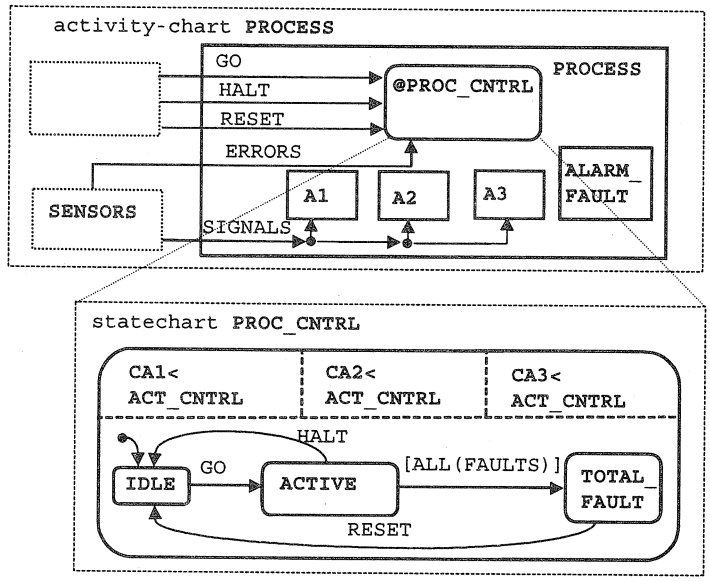
- When the formal parameter is of a user-defined type, the actual binding should also be of this user-defined type.
- Arrays must be of the same length and must have the same component types. If the index range of an array formal parameter is not specified, the index range values are inherited from the actual binding.
- Queues must have the same component types.
- Formal parameters cannot be defined directly as records and unions, because these structures are considered to be consistent only if they have the same user-defined type.

Note that because generic charts are the roots of the separate trees in the chart hierarchy, only elements appearing in global definition sets are commonly visible by them and to the charts of their instances. Therefore, user-defined types and constants that are used in the definition of the formal parameters must belong to some global definition set.

Finally, the bindings to ports must be consistent with the flow of information that appears in the activity-chart or module-chart of the instance. The binding to an `in` port should flow into the instance and the binding to an `out` port must be an output of the instance. This has indeed been adhered to in the example, as can be seen by inspecting Figs. 14.2*b* and 14.8.

14.4 Referring to Elements in Instances

An element that belongs to a generic chart will have an occurrence in the model for each instance of the chart. As explained earlier, instances of generic charts share elements with the rest of the model only via the parameters. In other words, it is impossible to refer to elements appearing in instances of charts outside the generic cluster, and therefore references to these elements do not appear in expressions of the model. However, testbenches, which do not obey *any* visibility rules (see Sec. 12.4.2), should be allowed to refer to elements in generic instances for purposes of analysis. In addition, external tools, such as



State: **CA1<ACT_CNTRL**
Defined in Chart: **PROC_CNTRL**
Actual Bindings:

Name	Type	Binding
GO	Event	GO
HALT	Event	HALT
RESET	Event	RESET
ERROR	Event	ERROR_1
A	Activity	A1
FAULT	Condition	FAULTS(1)

State: **CA2<ACT_CNTRL**
Defined in Chart: **PROC_CNTRL**
Actual Bindings:

Name	Type	Binding
GO	Event	GO
HALT	Event	HALT
RESET	Event	RESET
ERROR	Event	ERROR_2
A	Activity	A2
FAULT	Condition	FAULTS(2)

Figure 14.9 State instances and actual bindings.

simulators and prototype generators, should also allow references to these elements. For example, such tools should be able to present the value of each instance of an element, and we must provide a way to do so.

Going back to the example in Fig. 14.2, the TICK event is local to the generic chart PROCESS_SIGNAL. It has five occurrences in PROCESS_SIGNALS, one in each instance PS1 through PS5. Each occurrence can be identified by its instance name (e.g., PS1^TICK, which means "the element TICK in the instance PS1"). When the element name is not unique in the generic cluster, the chart name should be added to the element name. For example, if there is another TICK event in the subchart COMPUTE of Fig. 14.5, PS1^COMPUTE:TICK is how we would refer to the occurrence of COMPUTE:TICK in PS1, which is different from PS1^PROCESS_SIGNAL:TICK. The situation becomes more complicated when generic instances are nested within other generic charts, resulting in a chain of instance names (e.g., PS1^CMP3^X, which is "the element X in the generic instance CMP3 in the generic instance PS1"). Box names that might not be unique in their charts are identified in these references by unique pathnames. See App. A.1.