Chapter

3

Information Elements

This chapter deals with the information elements of our languages: events, conditions, and data-items. The data-items can be of simple *predefined types* or compound *user-defined types*. All information elements are defined in the Data Dictionary, and they can be used in both graphical charts and textual constructs. Each is defined as belonging to a particular chart of a global definition set, that is, to one of the specification components that make up the entire model. Information elements obey certain scoping rules that are described in Chap. 13.

3.1 Information Elements in the Model

The interface of the entire system, as well as that of each component, is an essential part of the specification and design, capturing the way it communicates with its environment. In many methodologies (with the exception of object-oriented design methods), a major part of the interface consists of a set of information elements that flow to and from the system or component. Very often the system development starts with the interface already given, and the specifier has to construct the model accordingly.

The interface specification must fit the nature of the system under description and its environment. For example, if the system communicates with a hardware environment, the interface may be specified in terms of bits in a connector structure. In communication systems, the interface description may consist of a message structure, sometimes adhering to an industrial standard or a predefined protocol. The information modeling can be on a very concrete level—listing the bits of the connector or computer word—or on a higher level—involving abstract events, conditions, and data-items. While the modeler in our approach is encouraged to use abstractions, a *bit* is supplied as one of the predefined types.

For example, assume that one of the functional components of the EWS is the operator panel driver, through which the OPERATOR inserts the commands and the range limits. The driver interprets the OPERATOR input and conveys it to the appropriate activity. The operator panel consists of the following components:

- Three command buttons:
  - *Set-up* For starting the setup procedure
  - *Execute* For starting the execution mode
  - *Reset* For transforming the system into an idle mode
- Ten digit keys, *0* to *9,* for entering the range limits
- An *Enter* key for indicating the entry end of a range-limit value
- A *Sensor Connected* switch for indicating that the sensor is connected

These elements can be represented on various levels of abstraction. The three commands can be referred to as events, or, alternatively, they can be three-bit data-items. The range limits can be modeled by a bit-array of ten bits presenting the ten digits sent one at a time or by whole numeric values and so on. To some extent, the choices depend on whether such decisions have already been made (i.e., whether the interface is given or is awaiting the design or implementation phase).

As another example, the fault report of the EWS is basically a textual report consisting of the following information components:

- The *time* when the fault occurred
- The *out-of-range value,* which is the computed value after the processing
- The *legal range limits*

Again, different levels of abstraction can be used here, depending on where the borders of the specification are placed. The fault report can be modeled as a string of limited length, as an array of strings—one for each line in the report—or as a record of the numeric values that specify the report contents without being too precise about the implementation details.

Information elements are used not only in specifying interfaces but in the detailed behavioral and functional specification. It is natural to use them to describe the logic and control of algorithms and to specify computations, just as variables are used in programming languages.

It is only natural to translate the requirement that the system checks if the value of the result of the processing is within the specified range to a construct that contains a condition expression such as:

```
(SAMPLE > LEGAL_RANGE.LOW_LIMIT) and
(SAMPLE < LEGAL_RANGE.HIGH_LIMIT)
```

Here, SAMPLE denotes the processed value, and the allowed limits of the range are captured by the two fields of the record LEGAL_RANGE. All these elements are conventional real values, and they can be compared using standard relation symbols such as < and >.

In our notation, information elements can appear along the flow-lines of activity-charts and module charts, and in the textual constructs used in behavioral and detailed functional descriptions. Information elements appear in reactions, triggers, and actions, and in other expressions in statecharts, mini-specs, and combinational assignments, as well as in the parameters of generic charts. We will see examples in the coming chapters.

Information elements and user-defined types are defined in the Data Dictionary, where their type and structure are specified. The names follow the naming rules of App. A.1. As for all kinds of elements appearing in the Data Dictionary, we may attach additional information to these elements, such as synonyms, textual description and user-defined attributes, using the standard mechanisms of the Data Dictionary. Some examples are given in the more detailed sections that follow. We can also use information elements whose values depend on other elements. Actually, these are named expressions, like macros and aliases in conventional programming, and they are also defined using the Data Dictionary.

The following sections describe the particular types of information elements and the user-defined types. The way these elements are used in behavioral descriptions will be discussed in Chaps. 4–8, particularly Chap. 5.

3.2 Events

Events are communication signals that indicate that something has happened. Very often they are used for synchronization purposes. When they flow they do not convey any content or value, only the very fact that they have occurred. They are thus instantaneous, and if not immediately sensed, they are lost.

In the EWS example, the activity COMPARE sends the event OUT_OF_RANGE to the control activity (through a control flow-line) to indicate that the tested value is not in the expected range. This event is an indication to the control activity that it should start its response to a fault occurrence, that is, posting an alarm and issuing a fault message.

Events are used extensively in the modeling of real-time systems to indicate interrupts, clock ticks, timing, and synchronization signals and to model cause/effect connections between different parts of the

system. In communication protocol modeling they mark message sending and acknowledge arrival. Events are also used in the modeling and implementation of interactive systems. Graphical user interface systems (GUIs) are based on user-generated occurrences and their subsequent responses and attached callbacks, all of which can be mapped naturally into events and corresponding reactions in our languages. This can be accomplished in a low-level fashion, referring to mouse button clicks and motions and keyboard manipulations or on a higher level, by abstracting them into menu selection and command activation.

In the EWS example, the OPERATOR's commands, EXECUTE, SET_UP, and RESET, are defined as events that control the system's operation. Here we chose the names to be imperative verbs, but it is also useful to use short phrases in the past tense for event names, such as OPERATION_COMPLETED or BUTTON_PRESSED.

In object-based decomposition, where the functional components consist of entities (or actors) and their associated operations, events can implement the request for individual operations. For example, we may model the request from the FAULT_HANDLER to DISPLAY_FAULT (i.e., post an alarm and issue a fault message) by an event bearing the same name; similarly, the event PRINT_FAULT will invoke the PRINT_FAULT operation.

A set of similar events can be organized in an array structure. For example, the EWS operator keyboard contains ten keys for digits, which are used to enter the range limits. The events of pressing these keys can be grouped in an array DIGIT_PRESSED consisting of ten event components. The individual component is accessed by its index in the array, just like in conventional programming languages, DIGIT_PRESSED(1) through DIGIT_PRESSED(10), where 10 stands for the digit 0. Chapter 5 shows how to detect that one of these ten events has occurred, without referring to each one explicitly. Figure 3.1 shows the Data Dictionary entry defining the event array. It shows that in addition to the array size designation we can also incorporate a short description and a long description, as in other Data Dictionary entries.

Other aspects of events, namely, event expressions and named event expressions, are discussed in Chap. 5, where our expression language is described in full.

3.3 Conditions

As with events, conditions are also used for control purposes. Conditions are persistent signals, that is, ones that hold for continuous time spans. They can be either true or false.

An example of a condition in the EWS is the signal SENSOR_CONNECTED, which is generated by the OPERATOR and is sensed by

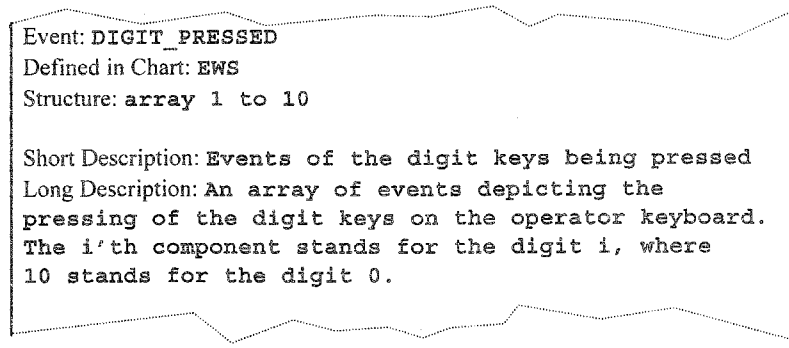


Figure 3.1 An event array in the Data Dictionary.

the control activity. This condition is self-explanatory, and it indicates whether the SENSOR is connected to the system—an essential prerequisite to activating the signal processing. Here, it is beneficial to use short phrases in the present tense as names of conditions to describe a situation that holds currently and for some continuous period of time.

Conditions are often used to describe the status of two-state entities, as in the preceding example. For example, a switch can be modeled by a condition SWITCH_ON that is either true or false. Conditions are also used to "remember" that some event has occurred until the required response is given.

Conditions, like events, can be organized in arrays to model the status of several similar elements. The information on the array index range is specified in the Data Dictionary.

Conditions, like other information elements, participate in detailed behavioral and functional descriptions. In subsequent chapters we shall see how they are manipulated, how they change values, and how they can influence the flow of control.

3.4 Data-Items

A data-item is a unit of information that may assume values of various types and structures. Data-items are very similar to the data elements in conventional programming languages: variables, constants, and so on. They maintain their values until they are explicitly changed and assigned new values.

Data-items are defined via the Data Dictionary, where their type and structure are specified, and other descriptive information can be added (e.g., attributes such as units, resolution, or distribution). Data-items can be of predefined types (integer, real, string, etc.), or records and unions composed of fields of various types. They can also be structured in arrays or queues. The modeler can also construct user-defined types

that are based on predefined types and structures. These concepts are described in the following sections.

3.4.1 Data-items of predefined types

The basic types of data-items are similar to those existing in programming languages. A data-item can be *numeric,* either *integer* or *real.* For example, in the EWS, the data-item `SAMPLE`, which is the result of the processing performed by the `PROCESS_SIGNAL` activity, has a numeric value and can be specified as real or integer. The value of an integer data-item is usually limited by 2^{31}. It is also possible to limit the values of an individual integer data-item by restricting its range, or by shortening its actual length (in bits). For example, if the EWS is extended to deal with five sensors, the identification number of a sensor will be an integer whose value will be restricted to the range 1 to 5. There is no limitation on real values.

When dealing with hardware systems, such as integrated circuits, it is natural to speak in terms of bits and bit-arrays. For this purpose, it is possible to define a *bit* data-item that can take on the values 0 and 1 or a *bit-array* data-item that consists of a sequence of bits. The definition of a bit-array data-item specifies its index range (which determines the number of bits) and direction, `to` or `downto`, which determines the most significant bit in its value. The index range limits are nonnegative integers.

In the EWS example, the sensor is a hardware component whose output, the `SIGNAL`, is described as a bit-array data-item. See Fig. 3.2. The signal consists of 24 bits, with bit 23 being the most significant. The syntax for such data-type expressions is described in App. A2.5.

Both bit and bit-array data-items are considered numeric, in the sense that they can participate in numeric expressions with no need of any explicit conversion, as discussed in Chap. 5. Values of bit-arrays are usually displayed in binary (e.g., `0B00101111`), octal (`00057`), or hexadecimal (`0X2F`), with the most significant bit being the leftmost one. A particular bit in the bit-array can be referred to explicitly. For example, `SIGNAL(23)` is the most significant bit of the sensor's output. Similarly, one can refer to a bit-array slice, such as `SIGNAL(2..0)`, which are the three bits of least significance. Note

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.

that if a bit-array is defined in the `to` (respectively, the `downto`) direction, the index range of its slices must be in ascending (respectively, descending) order.

A data-item can also be of type *string,* denoting a string of characters. String data-items are used when alphanumeric characters are involved, as in the EWS's `FAULT_REPORT`.

A string data-item can be used to introduce enumerated values. For example, we may define a string data-item `COMMAND` with one of three possible values, `'execute'`, `'set-up'`, or `'reset'`, that can be issued by the operator. Notice that the string value is written between single quotation marks. If needed, it is possible to specify the string length. For example, a data-item denoting an identifier name limited to 32 characters will be specified in the Data Dictionary with "Data-type: `string length=32`".

3.4.2 Records and unions

In addition to the basic types, a data-item can be a composition of named components, referred to as *fields,* each of which may be a data-item of any type or a condition. We support two kinds of compositions: *records* and *unions.* In a record, all components are present at any time, while a union contains, in any given time, exactly one of the components. Thus, a record can be viewed as an AND cluster of data, and a union as an OR cluster. The entire construct, record or union, is referenced by its name (e.g., on a flow-line), while a particular field is referenced using the dot notation:

```
record/union reference.field reference.
```

We mentioned that the `LEGAL_RANGE` data-item in the EWS is a record composed of two real fields: `LOW_LIMIT` and `HIGH_LIMIT`. The definition of this data-item in the Data Dictionary is shown in Fig. 3.3. The fields of `LEGAL_RANGE` are referenced by `LEGAL_RANGE.LOW_LIMIT` and `LEGAL_RANGE.HIGH_LIMIT`. The array notations and dot notation can be combined, so that if, for example, one of the fields of a record `R` is the bit-array `BA`, we may refer to the particular bit `R.BA(2)` or to the slice `R.BA(1..3)`.

A union construct is used when different types of values are relevant to different situations. For example, a union is useful when specifying a communication protocol that involves several kinds of messages, each carrying a different type of data.

Assume that the operator's input in the EWS example arrives via a single communication line that transfer two types of messages, commands and data (e.g., the range limits). Assume also that there is a channel along which the system is told the type of the arriving message. The data-item `MESSAGE_DATA` that carries the data can be

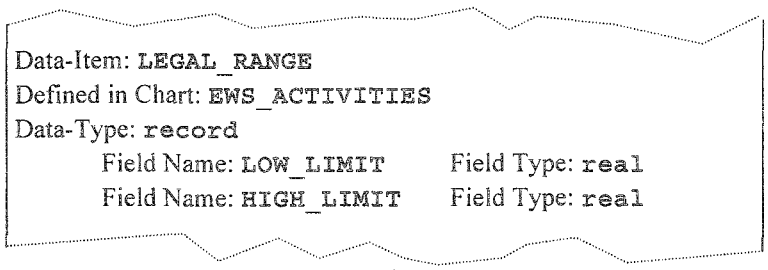


Figure 3.3 A record in the Data Dictionary.

defined to be a union of two possible fields: `COMMAND` of type *string* (see Sec. 3.4.1) and `LIMIT_VALUE` of type *real*. The system will refer to `MESSAGE_DATA.COMMAND` when it expects a string denoting the command, and to `MESSAGE_DATA.LIMIT_VALUE` when it expects the numeric range limit value. As explained before, at any given moment only one field of the union "exists," and it is illegal to refer to any other.

The field type attached to every field of the record or the union in the Data Dictionary can be of the following data-types: basic predefined types (e.g., integer, real, etc.; see the preceding section); condition, array, or queue (see the following section); or a user-defined type. The field cannot be defined to be another record or union; this kind of construction must be done with an intermediate definition of a user-defined type. See App. A2.5 for the syntax of data-type expressions.

3.4.3 Data-item structure

Data-items can be organized in structures—*arrays* or *queues*—with each component of the structure having one of the data-types described earlier or a user-defined type, as will be discussed shortly.

An array is a sequence consisting of a fixed predefined number of components. Assume, for example, that the EWS is enhanced to deal with five sensors. It is then natural to talk about an array of the sensor's signals: `SIGNALS`, defined as an array of five components, each of which is a bit-array, 23 `downto` 0. See Fig. 3.4.

Each array component can be of any of the basic predefined types, a record/union construct, or a user-defined type. Each component is accessed by its index (e.g., `SIGNALS(2)`), and double indexing is used to refer to components of components (e.g., `SIGNALS(1)(23)`). If the array component is a record, the dot notation can be combined with indexing. For example, if `AR` is an array of records that have two fields, `X` and `Y`, then we may use `AR(2).X` to access the `X` field of `AR(2)`.

The index range of the array is defined from *left index* `to` *right index*. There is no limitation on the array size. The index range limits are nonnegative integers, and the left index must be smaller or equal to the right index. (It might be more appropriate to call them "lower

index" and "upper index," but the names came from the range limits in bit-arrays.) It is very common to define an array going from 1 to some named integer constant (these are described in Chap. 5). Assume that we have a constant definition `NUMBER_OF_SENSORS = 5`. Then `SIGNALS` can be defined as `array 1 to NUMBER_OF_SENSORS`, to emphasize the fact that the size of the array depends on some other value.

Sometimes the size of one array depends on the size or index range of another. For example, we might want to set things up so that if the system allocates memory for an array, then any copy of it must be of the same size. In this case it is possible to use three predefined operators that apply to an array `V: length_of(V), lindex(V)`, and `rindex(V)`. The operators are evaluated to constant integer values.

A *queue,* as opposed to the fixed size arrays, is a dynamic list of components. Queues are described in detail in Chap. 8, where communication mechanisms are discussed. As in the case of arrays, the components of a queue can be of one of the predefined data types described earlier or a user-defined type. The components cannot be directly defined as records or unions; a queue of such components can be defined with an intermediate user-defined type. Queues are defined in the Data Dictionary just like the other data-items.

3.5 User-Defined Types

It is often the case that several data-items in the model have the same characteristics, such as their data-type. It can be useful to define a named data-type, called a *user-defined type,* that will be used to define them all. In addition to providing clarity, this reusability is also efficient because the full data-type definition appears in only one location in the Data Dictionary.

In the EWS example, the range construct, with the low and high limits, appears at least twice: in the current `LEGAL_RANGE` and in the `FAULT_REPORT` that contains the values against which the faulty processed signal was compared. We can have the Data Dictionary contain the definition of a user-defined type `RANGE`, which will be used later in the definition of these two data-items. This is shown in Fig. 3.5.

```
Data-Item: SIGNALS
Defined in Chart: EWS
Structure: array 1 to 5
Data-Type: bit-array 23 downto 0

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

Figure 3.4 An 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.

```
Data-Item: SCREEN
Data-Type: array 1 to 300 of ROW

User-Defined Type: ROW
Data-Type: array 1 to 200 of PIXEL

User-Defined Type: PIXEL
Data-Type: bit-array 7 downto 0
```

Figure 3.6 A definition of multidimensional array.

User-defined types are specified in terms of predefined types, record/union constructs, or data structures (arrays and queues). It is also possible to define them as other user-defined types or as conditions or arrays of conditions.

The user-defined type mechanism can also be used to define complex types, with multiple-level structure. The data-item FAULT_REPORT presented in Fig. 3.5 is a record, two of whose fields, FAULT_TIME and FAULT_RANGE, are themselves records. To achieve this multilevel structure, we must use the intermediate data-types, TIME and RANGE. We do not allow the definition of a record with an explicit record field.

There are no limitations on the multilevel usage of user-defined types. We can define multidimensional arrays, arrays of records, records with array fields, queues of records, etc., with any number of nesting levels.

For example, to specify a display screen whose size is 200 × 300 pixels, each of 8 bits, we use the data-item `SCREEN` and the user-defined types `ROW` and `PIXEL`, as shown in Fig. 3.6. A particular bit can be accessed by indexing. For example, `SCREEN(7)(2)(0)` is bit 0 in position (7,2) on the screen (i.e., pixel number 2 in row number 7).

In Chap. 13 we discuss the scope of elements (e.g., how their visibility depends on the chart in which they are defined). User-defined types are often required to be visible throughout the entire model, so they are usually defined in a global definition set, as discussed in Sec. 13.5.