Chapter

15

Related Approaches

In the preceding chapters we described the STATEMATE approach for modeling reactive systems. Other authors have also proposed methods for this purpose. Some have been presented as modeling languages, some as methodologies, and some as development standards. By and large, they address the same development stages and the same kind of systems that we do.

This chapter discusses a number of these methods. They are compared to the STATEMATE approach, and when appropriate we show how they can be used in conjunction with STATEMATE.

15.1 An Overview of Specification Methods

In this chapter we review methods that deal with reactive systems and cover the early stages of development, that is, specification and top-level design. We should add that by *methods* we mean methodologies, modeling languages, or standards.

A methodology is usually proposed together with a particular modeling language that matches its concepts. Similarly, modeling languages are more fitting for use with some methodologies than with others. The borderline between a methodology and a modeling language is thus not clear, and people often do not distinguish between the two. Our interest is mainly in methods that are heavily based on diagrammatic modeling.

We also discuss development standards. Standards are used in organizations, and are often imposed on modelers and system developers. In some sense they are similar to methodologies: They define processes of development, the various steps these processes involve, and their deliverables (such as special documents in predefined formats). Some standards are strict and very well defined, but some are more flexible.

The flexible ones—unlike most methodologies—do not dictate any particular modeling language, although they may talk about the concepts that should be dealt with when adhering to the standard and the resulting elements and relationships that should be described during the specification.

It is interesting to survey some of the highlights of methodologies for reactive systems. Our purpose is to represent the trends, and not to give a full history survey. Moreover, we concentrate only on methods that we found meaningful in comparison with STATEMATE.

Two variations of modeling methods based on the Structured Analysis paradigm (SA; see, for example, DeMarco 1978) were developed by Ward and Mellor (1986) and by Hatley and Pirbhai (1987). These authors added real-time provisions to the data-flow diagrams of the basic SA approach. The resulting extensions are usually referred to as *RTSA methods* (for real-time Structured Analysis).

A consolidation of these two methods into a new notation called ESML (Extended Systems Modeling Language) has been proposed by Bruyn et al. (1988). We discuss the Ward/Mellor and Hatley/Pirbhai methods in Sec. 15.2.

Around the same time, several other methods were developed for modeling reactive systems, taking a variety of approaches to the description of concurrent processes and their communication. Some examples are the Jackson System Development (JSD) method (Jackson 1983; Cameron 1989), Alford's Software Engineering Requirement Methodology/Distributed Computer Design System (SREM/DCDS) method (Alford 1985), and the CCITT Specification and Description Language (SDL) (International Telecommunication Union 1995). The last of these, SDL, was developed mainly for telecommunication systems, and it has evolved over the years into an object-based version. It is discussed in more detail in Sec. 15.4, which deals with other object-based methods.

Since the late 1980s, there has been an increasing interest in object-based and object-oriented techniques. This trend started in the programming community with the advent of object-oriented programming languages but has moved up to the earlier stages of specification and design, too. This is a natural process, because people want to avoid discontinuity between early and later stages of system development and to allow more natural traceability between them.

Several object-oriented methodologies and modeling languages have been proposed in the last few years, and some are aimed at reactive systems. We discuss three. The first is Real-time Object-Oriented Modeling (ROOM), an executable modeling language supported by the computerized tool ObjectTime (Selic et al. 1994). The second is Unified Modeling Language (UML) (Rational 1997), a broad and general approach, combining elements from the Booch method (Booch 1994), the Object-Modeling Technique (OMT) (Rumbaugh et al. 1991), and

scenario-like use-cases (Jacobson 1992), with Statecharts at its heart. The third is a UML-consistent executable language set called XOM (Executable Object Modeling), which was co-developed by one of us and is supported by a computerized tool, Rhapsody (Harel and Gery 1997).

Many methods, including some of those already mentioned, are the result of wide efforts, sometimes spanning a number of large companies. They were aimed at easing the task of developing complex systems in a particular industry or across several industries. For example, the Embedded Computer Systems Analysis and Modeling (ECSAM) methodology (Lavi and Winokur 1989) has been evolving since the early 1980s to address the needs of the Israel Aircraft Industries (IAI) in its system development projects. It uses the STATEMATE languages for modeling.

During the past 20 years or so many standards have been written and approved by various organizations, providing guidelines and criteria to be used in system development activities. Some of these are documented in Dorfman and Thayer (1990a). One of the best known standards is the U.S. Department of Defense Military Standard 2167A (*Military Standard* 1988) and its successor MIL-STD-498 (*Military Standard* 1994). Like most other standards, DOD-STD-2167A does not require the use of a particular modeling language. In Sec. 15.5 we show how our languages can be used to apply this particular standard.

The remainder of this chapter is devoted to briefly describing the aforementioned methods. We compare them with our own approach, and when relevant show how they can be used in conjunction with STATEMATE.

15.2 Methods Based on Structured Analysis

In the early 1980s, two methods were proposed, extending the classical structured analysis of DeMarco (1978) with means for modeling reactive, real-time systems, in the form of timing and control information. The two approaches are very similar and have the same expressive power. We shall concentrate on the parts of these methods that address the functional and behavioral views as defined in this book. Both methods contain portions that deal with the architecture of a system, too, but as far as we know, these aspects are rarely used, and the popular implementations of these methods do not cover them.

15.2.1 Ward and Mellor

The Ward/Mellor method was initiated by a group at Yourdon, Inc., principally by P. Ward and S. Mellor, and was described in detail in their 1986 book (Ward and Mellor 1996). We base our discussion on that book, concentrating on what the authors call *the transformation schema.*

The transformation schema contains diagrams based on the data-flow diagrams of DeMarco (1978). The diagrams of Ward and Mellor (1986) contain nodes for *data transformations* and for *control transformations* connected by edges depicting different types of flows between them. See Fig. 15.1*a*.

The *control transformations* are denoted by dashed circles. They map input event flows into output event flows. Among these are events whose effect is to *enable, disable,* or *trigger* a data transformation. The control transformations are described by state transition diagrams or tables. See Fig. 15.1*b* for a state transition diagram describing the control transformation in the transformation schema of Fig. 15.1*a*.

A *data transformation* may be stated procedurally in pseudocode or in any appropriate graphical or tabular language. The method also allows nonprocedural specification of relationships between the inputs and the outputs.

The notation for specifying data is a modified version of DeMarco's. Figure 15.2 presents the symbols used to define data compositions.

Ward and Mellor (1986) offer a way of executing the transformation scheme, which is based on the description of the execution of the Petri nets described by Peterson (1981).

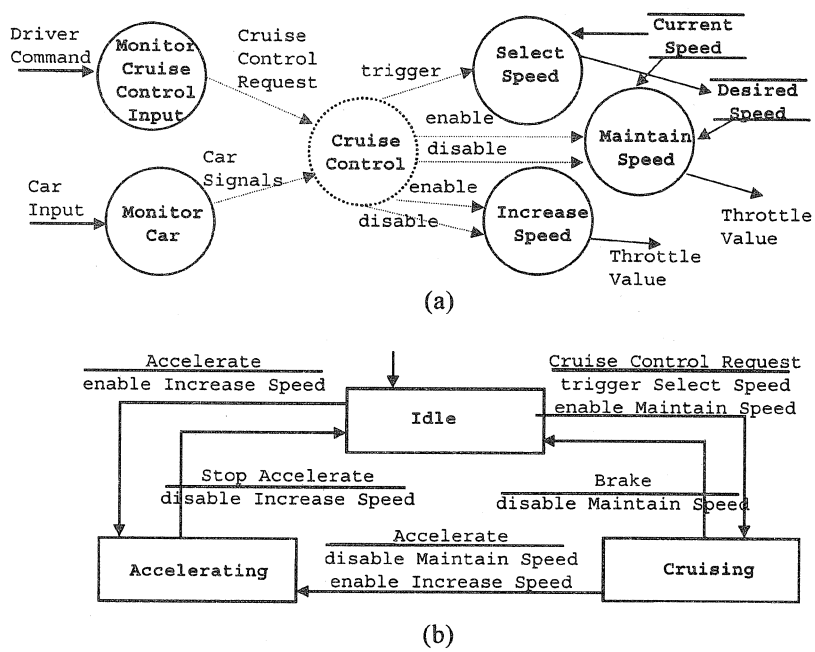


Figure 15.1 Diagrams in Ward/Mellor notation. (*a*) Transformation schema. (*b*) State transition diagram.

Symbol	Read as
=	is composed of
+	together with
[...|...|...]	select one of
m{ ... }n	at least m but no more than n iterations of

Figure 15.2 Data composition notation in the Ward/Mellor method.

15.2.2 Hatley and Pirbhai

The development of the Hatley/Pirbhai method was started by D. Hatley of Lear-Siegler, Inc., in collaboration with some engineers at Boeing. The method is described in detail in Hatley and Pirbhai (1987). Our discussion of the method is based on that book.

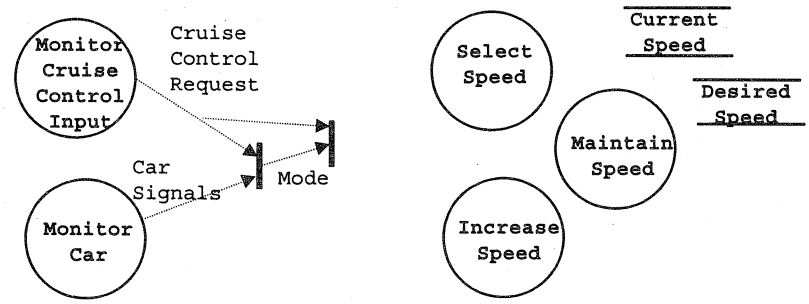
The Hatley/Pirbhai method prescribes that the system's requirements be captured by two related models: the *process model* and the *control model*.

The *process model* consists of a hierarchical structure of data-flow diagrams, each of which consists of processes describing the functions of the system, interconnected by data flows. A primitive process is described by a *process spec* (PSPEC). PSPECs are usually written in structured English, but they can also employ tables, diagrams, and equations. Each data flow is specified in *the requirement dictionary.*

The *control model* uses *control-flow diagrams* (which are very similar to the data-flow diagrams of the process model) to show the flow of control signals between the processes. Each nonprimitive process can be described by a pair consisting of a data-flow diagram and a control-flow diagram. The behavior of the process is described by *control specifications* (CSPECs), which are represented by a bar on the control-flow diagram, to show their input and output signals (their role is very similar to that of control activities in our Activity-charts). See Fig. 15.3*a*. The control specifications themselves may be presented in several ways: conventional state transition diagrams or tables, *decision tables* that describe functions between discrete inputs and outputs, or *process activation tables* that connect signal values with activation and deactivation of processes in a specified order. Figure 15.3*b* shows a process activation table.

Timing requirements can be added, too, specifying repetition rates of output signals in the requirement dictionary and input-to-output response times in tables or in timing diagrams.

The method does not include rigorous definitions of the languages used in PSPECs and CSPECs or in the timing requirements. However, the authors provide what they call balancing rules, which enable verification of a model's consistency.



(a)

INPUT		PROCESS		
Mode	Cruise Control Request	Select Speed	Increase Speed	Maintain Speed
idle	Off	0	0	0
	On	1	0	0
accelerating	Off	0	1	0
	On	0	1	0
cruising	Off	0	0	1
	On	0	0	1

(b)

Figure 15.3 Components of the Hatley/Pirbhai notation. (*a*) Control-flow diagram. (*b*) Process activation table.

15.2.3 Evaluation and comparison with STATEMATE

The Ward/Mellor and Hatley/Pirbhai methods are quite similar and have very similar expressive power. There are, however, some differences between them, especially with regard to the activation of processes. Both methods allow the use of a variety of languages for describing primitive processes, and both allow the use of different kinds of grammars and tables for this purpose. To implement these methods one must supply a rigorous syntax and semantics for whatever languages are used for this purpose.

As far as the relationship of these methods to ours is concerned, we note that all components of their languages have equivalents in ours. As both these methods use conventional state transition diagrams for control specification, they cannot take advantage of the features present in Statecharts, especially hierarchy, concurrency, history, and timing.

A significant deficiency of Ward/Mellor and Hatley/Pirbhai is their inability to deal with multiple similar components. There is no mech-

anism to deal with instances of a generic component, a feature that is essential for object-based modeling.

We refer the reader to the 1989 survey of Wood and Wood, which compares and evaluates the three approaches: those of Ward/Mellor, and of Hatley/Pirbhai, and ours (as well as a related forth one, ESML). This survey is quite illuminating, and it emphasizes the differences between the methods, particularly those relevant to modeling behavior. Davis' 1990 book contains interesting discussions and comparisons of these and other modeling approaches, too.

15.3 ECSAM

The Embedded Computer Systems Analysis and Modeling (ECSAM) methodology was developed at the Israel Aircraft Industries (IAI) for the analysis and design of computer-based systems (Lavi and Winokur 1989; Lavi et al. 1992). The method has evolved since the early 1980s, and it is used by several projects at the IAI. For modeling, it employs the languages described in this book. Some of the features of the structural view and its connection with the functional view that were described in Chaps. 9 and 10 were actually added to our languages to support the special needs of the ECSAM method.

According to ECSAM, a system is specified by two models—the *conceptual model* and the *design model.* Here we describe the conceptual model only. It consists of the following three views: the *logical modules view*, the *operating modes view*, and the *dynamic processes view*.

The *logical modules view* describes the partitioning of the system into logical subsystems (modules), the external information that flows between the system and its environment, and the information that flows between the subsystems. These are presented by module-charts, as described in Chap. 9. The logical modules view also defines the capabilities (activities) performed by each of the logical subsystems. This is done by linking an activity-chart that describes the module to each of the modules constituting the system, as explained in Sec. 10.2. In Fig. 15.4 the module-chart `SYSTEM` contains the system's logical modules. The capabilities of the logical module `M3` are described by the activity-chart `M3_ACTIVITIES`.

The *operating modes view* describes the main operating modes of the system and the transitions between them. This view is described by a statechart that is linked to the control activity of the entire system. In Fig. 15.4, the system's modes are described in the statechart `MODES`, which is connected to the control activity of the activity `PROCESSES`.

The *dynamic processes view* describes the behavioral processes that occur in the system in its various operating modes. This view is presented by a set of activity-charts. One activity-chart, which describes the system on the top level, details the processes (as activities) and connects them by the `throughout` construct to the states representing the

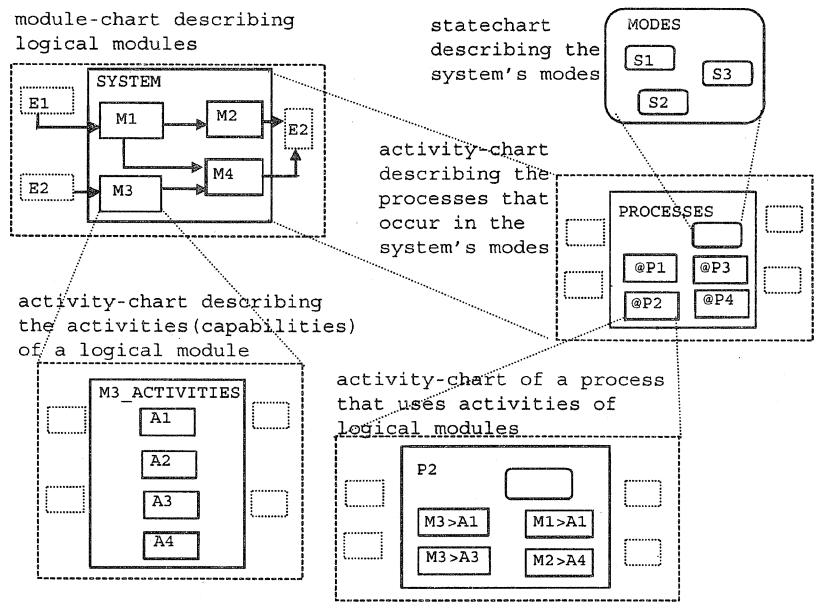


Figure 15.4 Relationships between ECSAM conceptual model components.

system modes (see Sec. 7.3.3). See the activity-chart `PROCESSES` in Fig. 15.4.

Each of the processes is then described by an off-page activity-chart containing the activities that constitute the process (e.g., see `P2` in Fig. 15.4). These activities are associated with the capabilities of the logical modules by the `is activity` relation, as explained in Sec. 10.4. For example, in Fig. 15.4 the activity named `M3>A1` is an occurrence of activity `A1` in module `M3`. The dynamics of the process is described by its control activity using a statechart.

In addition to defining these three views, the ECSAM method outlines roughly a dozen analysis steps that are to be applied to the system and to each of its subsystems. These can be found in Lavi et al. (1992).

15.4 Methods Based on Objects

We now discuss some methods that involve object-oriented concepts, such as abstract data types, object decomposition using class-instantiation techniques, and inheritance.

15.4.1 SDL

There have been several versions of Specification and Description Language (SDL) since its inception in 1976 as the Z.100 recommendation of CCITT. SDL was developed by the International Telecommunica-

tion Union mainly for telecommunication systems. Our review here is based on SDL-92 (International Telecommunication Union 1992), which extends SDL-88 by adding means for object-oriented modeling.

SDL is a rich language. It offers two different syntactic forms: a *graphic representation* (SDL/GR) and a textual *phrase representation* (SDL/PR), which are equivalent and are based on the same abstract grammar. The following example uses the graphical version.

A *system* in SDL is decomposed into *blocks* connected to each other and to the environment by *channels* that convey *signals.* The blocks are either further decomposed into other blocks or contain *processes.* A process in SDL is a kind of state machine that communicates with other processes or with the environment; processes are used to describe the behavior of the system.

A process, like any state machine, consists of *states,* in which it may consume signals. The *transition* between states is a sequence of *actions,* such as performing a *task* (assignment statement or informal text), making a *decision,* causing the *output of a signal,* setting a *timer,* calling a *procedure,* or creating an *instance of a process type.* See Fig. 15.5.

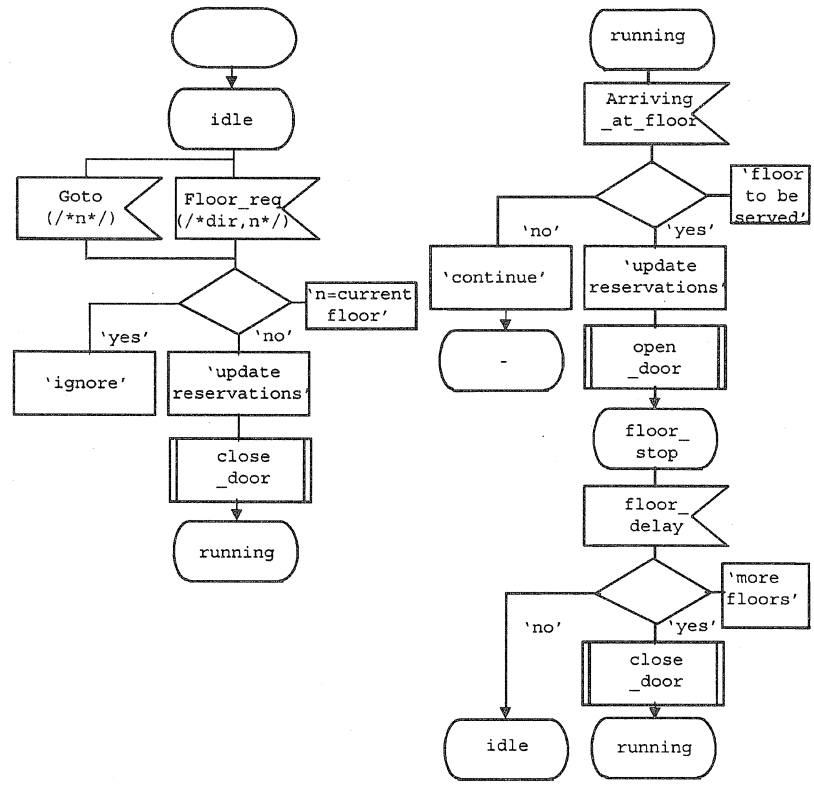


Figure 15.5 A process in SDL.

There are several ways in SDL to make the specifications more compact and easier to read. They include *referenced definitions, macros,* and *packages.* In particular, *types* of components are defined in packages that can be *used* in different contexts (e.g., in Ada), wherever the components are instantiated.

As a language, SDL is defined very rigorously. The recommendation document is very detailed (International Telecommunications Union 1995). It gives a formal definition—semantics included—for each entity and construct of the language. The processes—the dynamic components of the specification—are executable, with well-defined semantics.

As to the relationship to our languages, there are a many similarities, due to the similar evolution of both. They both support function-based and object-based decomposition. Their expressive power is quite similar. SDL, like our languages (and unlike the methods based on structured analysis, described in Sec. 5.2), has an instantiation mechanism, which is necessary in object-based modeling. SDL went one step further in support of object technology by including an inheritance (type specialization) mechanism. In contrast, our approach was to construct a separate language set and a separate tool specifically for object-oriented modeling; see Sec. 15.4.4.

15.4.2 ROOM

The Real-Time Object-Oriented Modeling (ROOM) language and methodology originated at the Telos Group at Bell-Northern Research in the late 1980s (Selic et al. 1994). This group started the development of the ObjectTime toolset, which supports the construction and execution of ROOM models.

The ROOM language is based on an object paradigm in which a system is viewed as a set of concurrently active objects, communicating by message passing. ROOM refers to these objects as *actors.* Each actor is an independent machine whose interface to its environment is defined by *ports.* Actors exchange messages through these ports. Each port has an associated *protocol* that restricts the type of messages that may flow through the port. Actors can be organized into a structure by connecting their ports via channels that are called *bindings.* A ROOM actor can itself be organized with an internal structure of component actors. See Fig. 15.6.

The behavior of an actor can be described by an extended state machine called a *ROOMchart,* which in the words of Selic et al. (1994) was "inspired by the Statechart formalism." Several features of Statecharts are included in ROOMcharts, such as state hierarchy (referred to as *composite states*), transitions exiting from the containing state (*group transitions*), condition connectors (*choicepoints*), and history entrances. On the other hand, ROOMcharts do not have

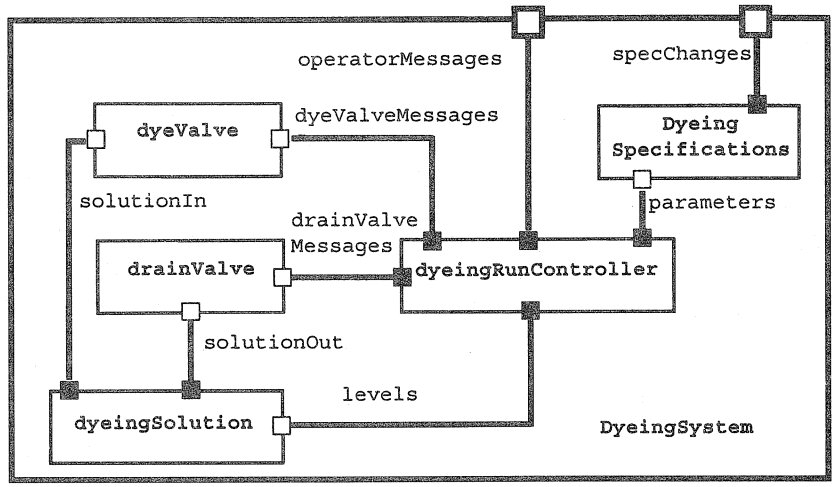


Figure 15.6 An actor in a ROOM model.

orthogonality; that is, they do not admit the and-state feature, a decision that is explained in Appendix C of Selic et al. (1994).

ROOM incorporates a conventional programming language into models (e.g., the ObjecTime implementation uses C++) to represent actions and data structures in low-level detailed descriptions.

ROOM models are based on class definitions for actors, protocols, and data objects. Class hierarchies are supported for each entity type, with different inheritance rules.

ROOM's concepts have a rigorous semantics, and its models are executable. The supporting tool, ObjecTime, includes a compiler that translates models into high-level source code that runs in a special run-time environment.

An interesting recent development is the newly announced commercial alliance between Rational Corporation (and their Rose toolset) and ObjecTime. This might effectively cause the ROOM method and the current ObjecTime tool to be dismissed and their development discontinued in favor of a UML-based approach.

15.4.3 UML

Unified Modeling Language (UML) is a large-scale effort to unify three of the many object-oriented methodologies that appeared in the late 1980s. These are the so-called Booch method (Booch 1994), the Object Modeling Technique (OMT) (Rumbaugh 1991), and Object-Oriented Software Engineering (OOSE) (Jacobsen 1992). The first two are general modeling methods that incorporate an object-based structural model, with *classes, object instances, relationships, aggregation, inheritance,* etc., and use variants of the Statecharts

language for modeling behavior. OOSE, on the other hand, is based on *use-cases.*

The unifying effort resulting in UML began in 1994 and is organized by Rational Corporation. It is led by the principal authors of the three aforementioned methods, G. Booch, J. Rumbaugh, and I. Jacobson. Many other people from several organizations participated in putting UML together, especially in the more recent effort on version 1.1, which was aimed at getting the earlier version 0.8 to be better defined. These include the people responsible for the ROOM method and its underlying tool ObjectTime (see Sec. 15.4.2) as well as the team responsible for i-Logix's object-oriented approach with its underlying tool, Rhapsody (see Sec. 15.4.4) represented by Eran Gery from I-Logix, Inc. and David Harel. UML version 1.1 was submitted as a proposal to the Object Management Group (OMG) Analysis and Design Task Force's RFP-1 for adoption as a standard. A decision by the OMG to adopt UML 1.1 as a standard was made in late 1997.

The UML involves many different kinds of diagrams. *Use-case diagrams* show the interaction of external entities with the system. These diagrams present the functional requirements of the system; they are similar in appearance to those in OOSE. *Class diagrams* are more or less the standard object models from the Booch method, OMT, and several other object-oriented methods. They show the collection of static model elements, their contents, and relationships. *Statechart diagrams* are based on the usual Statecharts, as defined here, with modifications that cater to object orientation. *Activity diagrams* are behavioral flow-chart–like diagrams. *Sequence diagrams* are a variant of MSCs (message sequence charts) found in many object-oriented writings. They show object interactions arranged in a time sequence. *Collaboration diagrams* also show object interactions, but they are organized around objects, and they show the relationships among them.

Detailed documents specifying the meta-model, notation, and semantics of UML can be found by following the links in www.rational.com/uml/ (Rational 1997). It is worth mentioning that one of the basic premises of UML is to leave many of the details vague enough to permit different implementations. This means that one can expect any number of tools to be built in the future, all claiming, correctly, to implement UML, although there might be quite significant differences between them.

15.4.4 XOM and Rhapsody

The Rhapsody tool is the first executable implementation of the core of the UML. It started out in the form of a carefully defined set of diagrammatic languages for modeling object-oriented systems with Statecharts at its heart, called XOM (Executable Object Modeling); see the conference version of Harel and Gery (1997). Joint work with the UML team has resulted in modifications to both approaches, so that

although the XOM language set of Harel and Gery (1997) does not cover all aspects of UML, it is fully compatible with it. In fact, this language set constitutes, in essence, the core executable portion of UML, and it comes complete with a fully worked-out behavioral semantics.

The XOM approach is supported by Rhapsody, a tool that enables model execution and full-code synthesis into object-oriented programming languages such as C++. The philosophy driving the development of Rhapsody is similar to the one that drove the development of STATEMATE, except that Rhapsody is used exclusively for object-oriented modeling, and it is intended more for software than for systems in general.

The XOM and Rhapsody approach involves two constructive modeling languages, *object-model diagrams* and *Statecharts,* and a reflective language, *message sequence charts* (MSCs, also called *sequence diagrams*). A language is *constructive* if it contributes to the dynamic semantics of the model. That is, its constructs contain information needed in executing the model or in translating it into executable code. Other languages are *reflective* and can be used by the system modeler to capture parts of the thinking that go into building the model—behavior included—or to derive and present views of the model to aid in analysis and to check for consistency against the constructive parts of the model. Object-model diagrams specify the structure of the system by identifying classes of objects (i.e., object types) and their multiplicities, object relationships and roles, subtyping, and inheritance. Especially noteworthy in object-model diagrams is the provision for specifying composite objects, which capture a strong form of aggregation; they are depicted by encapsulation, as in Activity-charts; see Fig. 15.7.

The behavior of an object is specified by a statechart that is associated with its class; see Fig. 15.8. Statecharts employ two mechanisms for interobject interaction, *events* and *operations.* An object can generate an event, which is queued, to be later consumed by the target object's statechart, and an object can also directly invoke an operation of another object, thus causing its statechart to carry out an appropriate method and perhaps return a value. One upshot of the hierarchical modeling of composite structure is that these interactions can be arranged to take on the form of either direct communication or broadcast. Statecharts can also create and destroy object instances and can delegate events to their components.

15.5 MIL-STD-498 (DOD-STD-2167A)

The system software development standard DOD-STD-2167A (*Military Standard* 1988) was used for many years by the contractors who developed software for the U.S. Department of Defense. A few years ago it was combined with the automated information system documentation

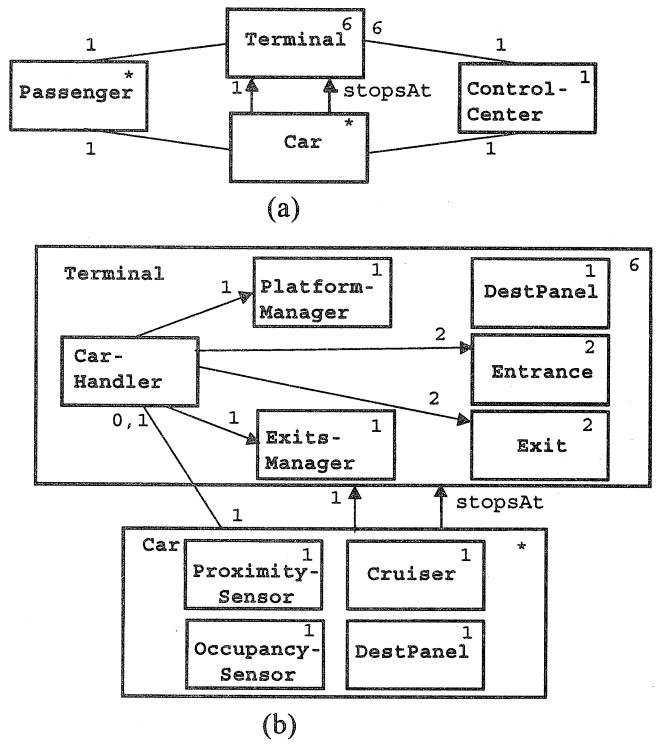


Figure 15.7 Object-model diagrams in a Rhapsody model. (a) High-level object-model diagram for a rail car. (b) Detailed diagram for composite objects.

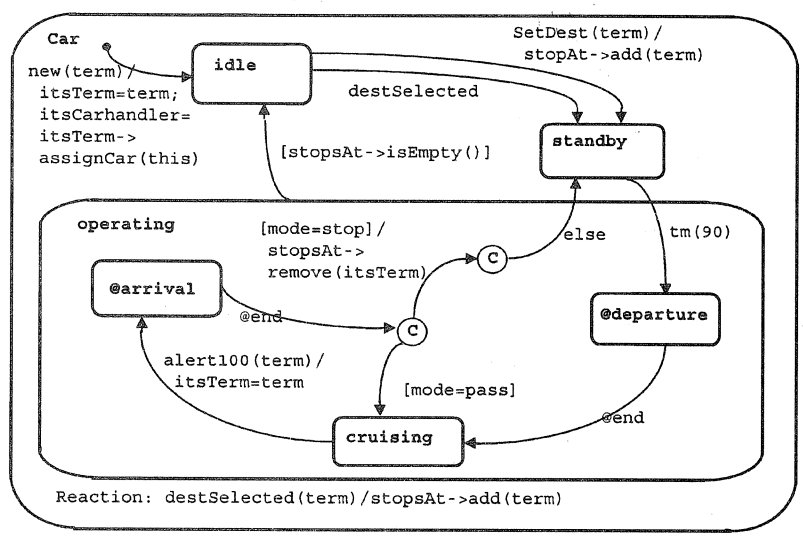


Figure 15.8 Statechart in a Rhapsody model.

standard DOD-STD-7935A to form the military standard MIL-STD-498 (*Military Standard* 1994). These standards detail the activities included in the software development process and provide a set of *data item descriptions* (DIDs), which form the requirements for documenting the development process.

Our languages, and hence the STATEMATE toolset, can be used to accomplish many of the specification and design activities required by these standards. The standards recommend using the DIDs as a checklist of items to be covered in the planning or engineering activity during the development and as a template for recording the results of this activity. Here we show how the concepts and terminology used in each DID of these standards map to the concepts and elements of our languages.

When conceptualizing the operational aspects of the system, a DID called the *operational concept description* (OCD) is used. The purpose of this phase is to obtain consensus among the acquirer, the developer, and the user on the operation of the system under development. Activity-charts and statecharts are used to describe the behavior of the system, and STATEMATE tools can be used to automatically generate a prototype of it.

During the system requirements phase the *system / subsystem specification* (SSS) and the *interface requirements specification* (IRS) are used. In this phase, an activity-chart describing the entire system (which is a module) is prepared, and the system capabilities are presented by the activities contained in it. The external and internal interfaces are represented by the flow-lines and their labeling data elements. These elements, in turn, are characterized in the Data Dictionary according to the requirements appearing in the DIDs. The system's states and modes, with their hierarchy, are described by a statechart, which is linked to the control activity of the top-level activity of the system.

During the system design phase the *system / subsystem design description* (SSDD) and the *interface design description* (IDD) are used. In this phase, the system components, which are the *computer software configuration items* (CSCIs) and *hardware configuration items* (HWCIs), are identified by a hierarchical module-chart, in which each component is represented by a module. The interfaces between the system components are described by the flow-lines between the modules. Each interface entity is a data element (information-flow, data-item, condition, or event) defined with all the required characteristics in the Data Dictionary.

During the software requirements phase the *software requirements specification* (SRS) and the *interface requirements specification* (IRS) are used. In this phase, a *computer software configuration item* (CSCI) is specified by an activity-chart whose activities present the CSCI's capabilities. The external and internal interfaces are described by the

TABLE 15.1 Mapping of MIL-STD-498 Documents to STATEMATE Concepts

Development Phase	Applicable IDDs	Modeling Constructs in the STATEMATE Languages
System requirements analysis	OCD	Activity-charts and statecharts describing the system behavior
	SSS IRS	Activity-chart and Data Dictionary describing the system capabilities and external interfaces. Statechart describing the system modes.
System design	SSDD IDD	Module-chart and Data Dictionary describing the CSCIs and HWCIs, and the interface data elements between these components
Software requirements analysis	SRS IRS	Activity-charts and Data Dictionary describing the CSCI's capabilities and the internal and external data elements
Software design	SDD IDD	Module-charts and Data Dictionary describing the CSCI's components (software units) and the interface data elements

flow-lines in this activity-chart, labeled by data elements that are defined in the Data Dictionary.

During the software (top-level) design phase the *software design description* (SDD) is used. In this phase, a CSCI is described by a module-chart. The CSCI components (software units) are defined as modules, which may be arranged in a hierarchy. The interface between these units is described by flow-lines between the modules labeled with the corresponding interface entities, accompanied by definitions in the Data Dictionary.

Table 15.1 presents STATEMATE modeling constructs that are used in each development phase according the standard.

An important facet of this connection between the military standard and the STATEMATE modeling languages is the fact that the same model can be used for all the development phases, with each phase refining the model and adding new parts. This assures good traceability between the phases, and makes it possible to document the traceability information together with the other specification details.

The STATEMATE system contains tools that support the production of the documents required by MIL-STD-498. Using these tools ensures that the documents are consistent. A description of this capability exists in the STATEMATE documentation provided by I-Logix.