Modeling Reactive Systems with Statecharts

Modeling Reactive Systems with Statecharts

The STATEMATE Approach

David Harel and Michal Politi

McGraw-Hill
New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

Library of Congress Cataloging-in-Publication Data

Harel, David, 1950–
Modeling reactive systems with statecharts : the statemate approach
David Harel, Michal Politi.
p. cm.
Includes index.
ISBN 0-07-026205-5
1. System design. 2. Computer software—Development.
3. Statecharts (Computer science) 4. Object-oriented programming
(Computer science) I. Politi, Michal. II. Title.
QA76.9.S88H3677 1998
005.1'2—dc21 98-17831
CIP

McGraw-Hill

A Division of The ***McGraw-Hill*** *Companies*

1 2 3 4 5 6 7 8 9 0 DOC/DOC 9 0 3 2 1 0 9 8

ISBN 0-07-026205-5

The sponsoring editor of this book was Simon Yates. The editing supervisor was Ruth Mannino, and the production supervisor was Clare Stanley. It was set in New Century Schoolbook by Kim Sheran and Paul Scozzari of McGraw-Hill's Professional Book Group Hightstown composition unit.

Printed and bound by R. R. Donnelley & Sons Company.

For our dear parents,

Joyce and Harold Fisch
and the late Zvi and Lena Frenkel

With love and respect

Contents

Preface xiii
Acknowledgments xv

Chapter 1. Introduction 1

1.1 System Development and Methodologies 1
1.1.1 Specification in a System Life Cycle 1
1.1.2 Development Methodologies and Supporting Tools 2
1.2 Modeling Reactive Systems 3
1.2.1 The Nature of Reactive Systems 3
1.2.2 An Example: The Early Warning System 4
1.2.3 Characteristics of Models 4
1.2.4 Modeling Views of Reactive Systems 5
1.2.5 Modeling Heuristics 7
1.3 The Modeling Languages 8
1.3.1 Activity-Charts 9
1.3.2 Statecharts 9
1.3.3 Module-Charts 11
1.3.4 Relationships between the Languages 11
1.3.5 Handling Large-Scale Systems 13
1.4 The STATEMATE Toolset 14

Chapter 2. The Functional View: Activity-Charts 19

2.1 Functional Description of a System 19
2.1.1 Functional Decomposition 20
2.1.2 Function-Based Decomposition 21
2.1.3 Object-Based Decomposition 22
2.1.4 System Context 22
2.1.5 The Decomposition Process 23
2.2 Activities and Their Representation 24
2.2.1 The Hierarchy of Activities 24
2.2.2 The Context of an Activity 25
2.3 Flow of Information between Activities 26
2.3.1 Flow-Lines 26
2.3.2 Flowing Elements 29
2.3.3 Information-Flows 30
2.3.4 Data-Stores 31

2.4 Describing the Behavioral Functionality of Activities 33
2.4.1 Control Activities 33
2.4.2 Activities in the Data Dictionary 34
2.5 Connectors and Compound Flow-Lines 36
2.5.1 Joint Connectors (Fork and Merge Constructs) 36
2.5.2 Junction Connectors 37
2.5.3 Diagram Connectors 39
2.5.4 Compound Flow-Lines 39

Chapter 3. Information Elements 41

3.1 Information Elements in the Model 41
3.2 Events 43
3.3 Conditions 44
3.4 Data-Items 45
3.4.1 Data-Items of Predefined Types 46
3.4.2 Records and Unions 47
3.4.3 Data-Item Structure 48
3.5 User-Defined Types 49

Chapter 4. The Behavioral View: Statecharts 53

4.1 Behavioral Description of a System 53
4.2 Basic Features of Statecharts 55
4.3 The Hierarchy of States 58
4.4 Orthogonality 61
4.4.1 And-States and Event Broadcasting 61
4.4.2 Conditions and Events Related to States 63
4.4.3 Multi-Level State Decomposition 64
4.5 Connectors and Compound Transitions 65
4.5.1 Condition and Switch Connectors 65
4.5.2 Junction Connectors 66
4.5.3 Diagram Connectors 68
4.6 More about Transitions 68
4.6.1 Transitions to and from And-States 68
4.6.2 History Entrances 71

Chapter 5. The Textual Expression Language 73

5.1 Event, Condition and Data-Item Expressions 73
5.1.1 Event Expressions 74
5.1.2 Condition Expressions 75
5.1.3 Data-Item Expressions 76
5.1.4 Named Expressions 78
5.2 Actions 80
5.2.1 Element Manipulation 80
5.2.2 Compound Actions and Context Variables 82
5.2.3 Iterative Actions 84
5.3 Time-Related Expressions 86
5.3.1 Timeout Events 86
5.3.2 Scheduled Actions 86
5.4 Static Reactions 87

5.4.1 Reactions on Entering and Exiting a State 87
5.4.2 General Static Reactions 87

Chapter 6. The Semantics of Statecharts 91

6.1 Execution of the Model 91
6.1.1 External Changes and System Reactions 91
6.1.2 The Details of Status and Step 95
6.2 Handling Time 97
6.2.1 Sequentiality and Time Issues 97
6.2.2 Time Schemes 98
6.3 Nondeterministic Situations 99
6.3.1 Multiple Enabled Transitions 99
6.3.2 Racing 100

Chapter 7. Connections between the Functional and Behavioral Views 101

7.1 Dynamics in the Functional Decomposition 101
7.2 Dynamics of Activities 103
7.2.1 Statecharts in the Functional View 103
7.2.2 Termination Type of an Activity 104
7.2.3 Perpetual Activities 105
7.3 Controlling the Activities 106
7.3.1 Starting and Stopping Activities 106
7.3.2 Sensing the Status of Activities 108
7.3.3 Activities throughout and within States 108
7.3.4 Suspending and Resuming Activities 110
7.4 Specifying Behavior of Basic Activities 111
7.4.1 Reactive Minispecs 112
7.4.2 Procedure-Like Mini-Specs 113
7.4.3 Combinational Assignments 114

Chapter 8. Communication between Activities 117

8.1 Communication and Synchronization Issues 117
8.2 Controlling the Flow of Information 118
8.2.1 Elements Related to Flow of Information 118
8.2.2 Interface between "Execution" Components 120
8.3 Examples of Communication Control 121
8.3.1 Communication between Periodic Activities 122
8.3.2 Message Passing 123
8.4 Activities Communicating through Queues 124
8.4.1 Queues and Their Operation 125
8.4.2 The Semantics of Queues 126
8.4.3 Queues in an Activity-Chart 127
8.4.4 Example of Activities Communicating through Queues 128
8.4.5 An Address of a Queue 129

Chapter 9. The Structural View: Module-Charts 131

9.1 Structural Description: High-Level Design 131
9.2 Internal and External Modules 132

9.3 Communication Lines between Modules 134
9.3.1 Flow of Information between Modules 135
9.3.2 Physical Links between Modules 135
9.4 Connectors and Compound Flow-Lines 136

Chapter 10. Connections between the Functional and Structural Views 139

10.1 Relating the Functional and Structural Models 139
10.1.1 Functional Description of a Module 140
10.1.2 Allocating Activities to Modules 140
10.1.3 Mapping Activities to a Module's Activities 142
10.2 Activity-Chart Describing a Module 142
10.3 Activities Implemented by Modules 145
10.4 Activities Associated with a Module's Activities 148
10.5 Object-Oriented Analysis with Module-Charts 149

Chapter 11. Splitting Up Charts 153

11.1 Separating a Chart into Multiple Pages 153
11.2 Offpage Charts 155
11.3 Connecting Offpage Charts by Matching Flows 157
11.4 Connecting Offpage Statecharts Using Connectors 160

Chapter 12. Putting Things Together 163

12.1 Relationships among the Three Kinds of Charts 163
12.2 A Chart in a Model 166
12.3 Hierarchy of Charts 166
12.4 Entities External to the System under Description 167
12.4.1 Environment Modules of Activities 167
12.4.2 Testbenches 169

Chapter 13. Scope and Resolution of Elements 173

13.1 Visibility of Elements and Information Hiding 173
13.2 Defining, Referencing and Resolving Elements 174
13.3 The Scope of Charts and Graphical Elements 176
13.3.1 Referring to Charts and Box Elements 176
13.3.2 Referring to Activities in Statecharts 177
13.3.3 Referring to States in Statecharts 178
13.3.4 External Activities or Modules 178
13.3.5 Referring to Modules and Activities in Activity-Charts 179
13.4 The Scope of Textual Elements 180
13.4.1 Visibility of Textual Elements 180
13.4.2 Naming Textual Elements 181
13.4.3 More about Resolution of Textual Elements 183
13.5 Global Definition Sets (GDSs) 184

Chapter 14. Generic Charts 187

14.1 Reusability of Specification Components 187

14.2 Definition and Instances of Generic Charts 189
14.2.1 Notation and Basic Rules of Generics 189
14.2.2 Generic Charts in the Chart Hierarchy 191
14.3 Parameters of Generic Charts 192
14.3.1 Formal Parameters of a Generic Chart 193
14.3.2 Actual Bindings of Parameters 196
14.4 Referring to Elements in Instances 197

Chapter 15. Related Approaches 201

15.1 An Overview of Specification Methods 201
15.2 Methods Based on Structured Analysis 203
15.2.1 Ward and Mellor 203
15.2.2 Hatley amd Pirbhai 205
15.2.3 Evaluation and Comparison with STATEMATE 206
15.3 ECSAM 207
15.4 Methods Based on Objects 208
15.4.1 SDL 208
15.4.2 ROOM 210
15.4.3 UML 211
15.4.4 XOM and Rhapsody 212
15.5 MIL-STD-498 (DOD-STD-2167A) 213

Chapter 16. Transition to Design 217

16.1 STATEMATE Models in the Development Process 217
16.1.1 Models as Prototypes 218
16.1.2 Design using Specification Models 218
16.1.3 Restructuring for Design 219
16.2 Mapping Models to Design Structures 219
16.2.1 Design Criteria 219
16.2.2 Evaluation of the Mapping 221

Appendix A. Names and Expressions 225

A.1 Names 225
A.1.1 Reserved Words 225
A.1.2 Textual Element Names 225
A.1.3 Box Element Names 226
A.1.4 Names of Elements in Generic Instances 226
A.2 Expressions 227
A.2.1 Event Expressions 227
A.2.2 Condition Expressions 228
A.2.3 Data-Item Expressions 230
A.2.4 Action Expressions 233
A.2.5 Data-Type Expressions 234
A.3 Predefined Functions 235
A.3.1 Arithmetic Functions 235
A.3.2 Trigonometric Functions 235
A.3.3 Random Functions 236
A.3.4 Bit-Array Functions 236
A.3.5 String Functions 236

A.4 Reactions and Behavior of Activities 237
A.4.1 Statechart Labels 237
A.4.2 State Reactions and Reactive Mini-Specs 237
A.4.3 Procedure-Like Mini-Specs 237
A.4.4 Combinational Assignments 237
A.5 Flow of Information 237
A.5.1 Flow Labels and Information-Flow Components 237
A.5.2 Actual Bindings of Generic Parameters 238

Appendix B. Early Warning System Example: Functional Decomposition Approach 239

B.1 Textual Description of the System 239
B.2 The Model 239
B.2.1 The Hierarchy of Charts 239
B.2.2 The Charts 240
B.2.3 The Data Dictionary 243

References 251

Index 253

Preface

This book provides a detailed description of a comprehensive set of languages for modeling reactive systems. The approach is dominated by the language of Statecharts, which is used to describe behavior, combined with Activity-charts, which are used for describing the system's activities (i.e., its functional building blocks, capabilities, and objects) and the data that flows between them. These two languages are used to develop a conceptual model of the system, which can be combined with the system's physical, or structural, model described in our third language, Module-charts. These three languages are highly diagrammatic in nature, constituting full-fledged visual formalisms, complete with rigorous semantics. They are accompanied by a Data Dictionary for specifying additional parts of the model that are textual in nature.

The approach described here lies at the heart of the STATEMATE system, which the authors have helped design and build at I-Logix, Inc. since 1984. STATEMATE is most beneficial in requirements analysis, specification, and high-level design. In addition to supporting the modeling effort using the aforementioned language set, STATEMATE provides powerful tools for inspecting and analyzing the resulting models, via model execution, dynamic testing, and code synthesis.

This book discusses the modeling languages in detail, with an emphasis on the language of Statecharts, because it is the most important and intricate language in the set and the most novel. Statecharts are used to specify the behavior of activities, whether they represent functions in a functional decomposition or objects in an object decomposition. We describe the syntax in a precise and complete manner and discuss the semantics in a way that is intended to render the model's behavior clear and intuitive. Our presentation is illustrated extensively with examples, most of which come from a single sample model of an early warning system (EWS). Appendix B provides a summarized description of this model.

Whenever possible, we have tried to explain our motivation in including the various features of the languages. We also provide hints and guidelines on such methodological issues as decomposition criteria and the order in which charts are to be developed.

While we do provide a brief description of the STATEMATE system in Sec. 1.4, this book is not intended to be a user manual for it but, rather, a definitive description of its languages and a guide to their use. For more on STATEMATE's capabilities, we refer the reader to the documentation supplied by I-Logix, Inc.

This book should be of interest to a wide variety of systems developers (both in software and hardware) and to teachers and students of software and hardware engineering.

Acknowledgments

Thanks are due to Jonah Lavi for initiating David Harel's interest in this area in 1983, an interest that led to the invention of the Statecharts language. The other two languages, Activity-charts and Module-charts, and the ways they are integrated with Statecharts, were developed by the authors together with several people, predominantly Rivi Sherman and Amir Pnueli. We would also like to express our deep gratitude to the many other extremely talented and dedicated people at I-Logix Israel, Ltd., led with insight, wisdom, and sensitivity by Ido and Hagi Lachover, for conceptualizing, designing, and building the STATEMATE tool.

Chapter

1 Introduction

This chapter describes the role of models in a system development life cycle and characterizes reactive systems, the ones for which the languages of this book are particularly suited. It then introduces the early warning system (EWS), a reactive system that we shall use as a running example throughout the book. It also presents an overview of the modeling languages and a brief description of the STATEMATE toolset, which was built around the language of Statecharts and which supports the modeling process and provides means for executing and analyzing the models, synthesizing code from them, and more.

1.1 System Development and Methodologies

We first describe the background for our work and the context in which our modeling languages fit.

1.1.1 Specification in a system life cycle

It is common practice to identify several phases in the development life cycle of a system, each of which involves certain processes and tasks that have to be carried out by the development team. The main phases of the classic waterfall model (Royce 1970) are *requirements analysis,* and *specification, design, implementation, testing,* and *maintenance.* Over the past 20 years, many variations of this model were proposed, as well as quite different approaches to the life cycle (Dorfman and Thayer 1990b). Some center around prototyping, incremental development, reusable software, or automated synthesis.

Most proposals for system development life cycle patterns contain a requirements analysis phase. Correcting specification errors and misconceptions that are discovered during later stages of the system's life cycle is extremely expensive, so it is commonly agreed that thorough comprehension of the system and its behavior should be carried out as early as possible. Special languages are therefore used in the requirements analysis phase to specify a model of the system, and special techniques are used to analyze it extensively. As described later, we advocate various kinds of analyses, including model execution and code synthesis. In this book, we shall use the terms *model* and *specification* interchangeably.

The availability of a good model is important for all participants in the system's development. If a clear and executable model is constructed early on, customers and subcontractors, for example, can become acquainted with it, and can approve of the functionality and behavior of the system before investing heavily in the implementation stages. Creating precise and detailed models is also in the best interest of the system's designers and testers. Clearly, the specification team itself uses modeling as the main medium for expressing ideas and exploits the resulting models in analyzing the feasibility of the specification. Chapter 16 contains more about the ways our models can be used for later stages of development.

1.1.2 Development methodologies and supporting tools

A term commonly used in connection with the development process is *methodology.* A methodology provides guidelines for performing the processes that comprise the various phases. Concentrating on the modeling and analysis phase, we may say that a methodology consists of the following components:

- The methodology's *underlying approach* and the *concepts* it uses, that is, the terms and notions used to capture the conceptual construct of the system and to analyze it.
- The *notation* used, that is, the modeling languages with their syntax and semantics. Sometimes these contain constructs that are sufficiently generic to be relevant to several different concepts of the underlying approach.
- The *process* prescribed by the methodology, that is, which activities have to be carried out to apply the methodology and in what order, how does the work progress from one activity to the next, and what are the intermediate outputs or products of each. The methodology usually also provides heuristics for making the process more beneficial.
- The computerized *tools* that can be used to help in the process.

This book is mainly about notation, in that it describes a set of modeling languages and illustrates their use. However, it also describes several concepts and notions that underly a number of development methodologies. Thus, while our approach to modeling and analysis is not necessarily connected to any particular methodology, it is more compatible with some methodologies than with others (just as flexible programming languages can be used with very different program design and implementation methods but might be more fitting for some specific ones). In particular, our approach can be used smoothly with variants of Structured Analysis (DeMarco 1978; Military Standard 1988) as well as with other methodologies, such as object-oriented analysis. Moreover, although the book does not get into the details of any particular methodological process, we do describe the STATEMATE set of tools (from I-Logix, Inc.) later in the chapter. STATEMATE can be used in conjunction with several relevant methodologies to apply our modeling and analysis approach, and implements all features of the languages described in the book.

1.2 Modeling Reactive Systems

As explained above, the heart of the specification stage is the construction of the system model. In this section we discuss the overall nature and structure of models, thus preparing for the subject matter of the book, which involves the modeling languages themselves. However, we should first say something about the kinds of systems we are interested in.

1.2.1 The nature of reactive systems

Our modeling approach, particularly the Statecharts language, is especially effective for reactive systems (Harel and Pnueli 1985; Manna and Pnueli 1992), the behavior of which can be very complex, causing the specification problem to be notoriously elusive and error-prone. Most real-time systems, for example, are reactive in nature.

A typical reactive system exhibits the following distinctive characteristics:

- It continuously interacts with its environment, using inputs and outputs that are either continuous in time or discrete. The inputs and outputs are often asynchronous, meaning that they may arrive or change values unpredictably at any point in time.[1]

[1]This should be contrasted with *transformational systems,* in which the timing of the inputs and outputs is much more predictable. A transformational system repeatedly waits for all its inputs to arrive, carries out some processing, and outputs the results when the processing is done.

- It must be able to respond to interrupts, that is, high-priority events, even when it is busy doing something else.
- Its operation and reaction to inputs often reflects stringent time requirements.
- It has many possible operational scenarios, depending on the current mode of operation and the current values of its data as well as its past behavior.
- It is very often based on interacting processes that operate in parallel.

Examples of reactive systems include on-line interactive systems, such as automatic teller machines (ATMs) and flight reservation systems; computer-embedded systems, such as avionics, automotive, and telecommunication systems; and control systems, such as chemical and manufacturing systems.

1.2.2 An example: The early warning system

Many of the characteristics mentioned earlier are present in the simple early warning system (EWS) that we use as an example throughout this book to illustrate the ideas and features of the languages. The EWS monitors a signal arriving from outside, checks whether its value is in some predefined range, and if not, notifies the operator by an alarm and appropriate messages. This is a general kind of system, the likes of which can be found in a variety of applications. Here is a brief informal description of the EWS that will become useful for understanding the details later on:

> The EWS receives a signal from an external sensor. When the sensor is connected, the EWS processes the signal and checks if the resulting value is within a specified range. If the value of the processed signal is out of range, the system issues a warning message on the operator display and posts an alarm. If the operator does not respond to this warning within a given time interval, the system prints a fault message on a printing facility and stops monitoring the signal. The range limits are set by the operator. The system becomes ready to start monitoring the signal only after the range limits are set. The limits can be redefined after an out-of-range situation has been detected or after the operator has deliberately stopped the monitoring.

See Fig. 1.1 for the schematic structure of the EWS.

1.2.3 Characteristics of models

A system model constitutes a tangible representation of the system's conceptual and physical properties and serves as a vehicle for the specifier and designer to capture their thoughts. In some ways, it is like the set of plans drawn by an architect to describe a house. It is used mainly

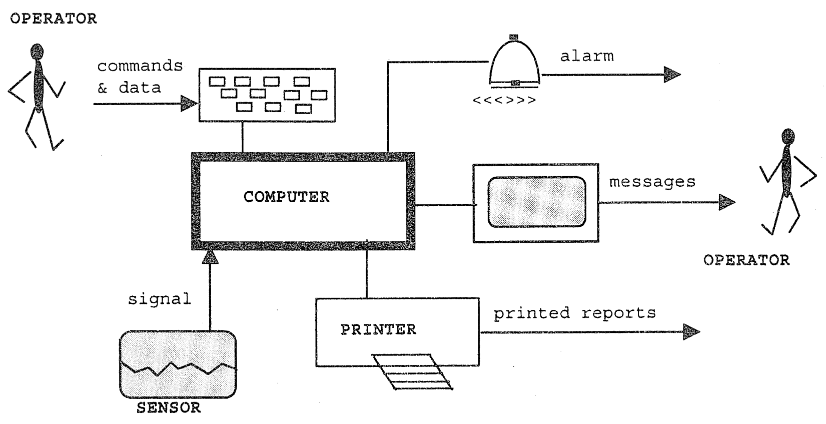


Figure 1.1 The early warning system (EWS).

for communication, but it should also facilitate inspection and analysis. The modeling process involves conceiving the elements relevant to the system and the relationships between them and representing them using specific, well-defined languages. When the model reflects some preexisting descriptions, such as requirements written in natural language, it is useful to keep track of how the components of the developing model are derived from the earlier descriptions.

To achieve the goal of enabling systems developers to model a system, our modeling languages have been designed with several important properties in mind: to be intuitive and clear, to be precise, to be comprehensive, and to be fully executable. To achieve clarity, elements of the model are represented graphically wherever possible; for example, nested box shapes are used to depict hierarchies of elements, and arrows are used for flow of data and control. For precision, all languages features have rigorous mathematical semantics, which is a prerequisite for carrying out meaningful analysis. Comprehension comes from the fact that the languages have the full expressive power needed to model all relevant issues, including the what, the when, and the how. As for executabilty, the behavioral semantics is detailed and rigorous enough to enable the model to be both executed directly, like a computer program, and to be translated into running code for prototyping and even for implementation purposes.

1.2.4 Modeling views of reactive systems

Building a model can be considered as a transition from ideas and informal descriptions to concrete descriptions that use concepts and predefined terminology. In our approach, the descriptions used to capture the system specification are organized into three views, or projections, of the system: the *functional,* the *behavioral,* and the *structural.* See Fig. 1.2.

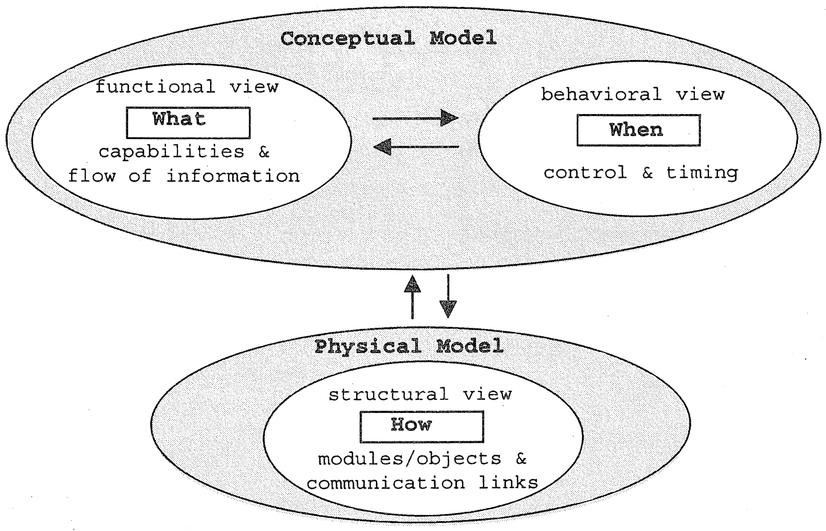


Figure 1.2 The three specification views.

The *functional view* captures the "what." It describes the system's functions, processes, or objects, also called activities, thus pinning down its capabilities. This view also includes the inputs and outputs of the activities, that is, the flow of information to and from the external environment of the system as well as the information flowing among the internal activities. For example, the activities of the EWS include sampling the input signal, comparing the read signal value with the predefined limits, and generating an alarm. The information flows in the EWS include the signal that flows from the external sensor, the operator commands that are input from the operator console, and the message and alarm notification that are output to the operator.

The *behavioral view* captures the "when." It describes the system's behavior over time, including the dynamics of activities, their control and timing behavior, the states and modes of the system, and the conditions and events that cause modes to change and other occurrences to take place. It also provides answers to questions about causality, concurrency, and synchronization. In the EWS example, the behavioral view might identify those states in which the system is waiting for commands, processing the signal, generating an alarm, or setting up new limit values. The behavioral view would also identify the events that cause transitions between these states. For example, it would specify what causes the system to generate an alarm or when the processing stops and the set-up procedure starts. Hence, it specifies precisely when the activities described in the functional view are active, and when the information actually flows between them.

There is a tight connection between the functional and behavioral views. Activities and data-flow need dynamic control to come to life, but the behavioral aspects are all but worthless if they have nothing to control. Technically, each activity in the functional view can be provided with a behavioral description given in the behavioral view, whose role it is to control the activity's internal parts, that is, its subactivities and their flow of information.

The *structural view* captures the "how." It describes the subsystems, modules, or objects constituting the real system and the communication between them. The EWS could be specified in the structural view to consist of an operator monitor, a control and computation unit, a signal processor, an alarm generator, and so on.

While the two former views provide the *conceptual model* of the system, the structural view is considered to be its *physical model,* because it concerns itself with the various aspects of the system's implementation. As a consequence, the conceptual model usually involves terms and notions borrowed from the problem domain, whereas the physical model draws more upon the solution domain.

The main connection between the conceptual and physical models is captured by specifying the modules of the structural view that are responsible for implementing the activities in the functional view. For example, the EWS activity that compares the input signal with the predefined limit values is implemented in the control and computation unit.

1.2.5 Modeling heuristics

Modeling heuristics are guidelines for how the notation should be used to model the system. This involves several issues, such as:

- The mapping between the methodology's concepts and the elements allowed in the notation. If the notation is flexible and its constructs can be used to depict several different concepts, this mapping has to be defined carefully.
- The type of decomposition to be used. Some possibilities are decompositions that are function based, object based, mode based, module based, or scenario based. The type chosen depends, in general, on the conceptual base of the methodology, although within a given methodology there is often some flexibility, according to the nature of the system and the role the model will play in the overall development effort. In the context of our notation, this issue is mainly relevant to the functional view and will be discussed further in Chap. 2.
- The step-by-step order of the modeling process. Which view are we to start with? Should we be working in a bottom-up or top-down fashion? Again, this is an issue that mostly depends on the methodology, but it is also affected by what is already known about the system.

In addition, modeling guidelines are often concerned with more marginal details, such as naming conventions and the number of allowed offspring in each decomposition level as well as layout rules for improving the model's aesthetics and clarity.

In this book we do not mean to address or recommend any specific global methodology. Although most parts of our running example will use a particular method, we will mention other possibilities, too.

1.3 The Modeling Languages

The three views of a system model are described in our approach using three graphical languages: *Activity-charts* for the functional view, *Statecharts* for the behavioral view, and *Module-charts* for the structural view. Additional nongraphical information related to the views themselves and their inter-connections is provided in a *Data Dictionary*. See Fig. 1.3.

Some of the basic ideas that make up our languages have been adapted from other modeling languages, such as data-flow diagrams, state-transition diagrams, data dictionaries and minispecs. However, they include many extensions that increase their expressive power and simplify and clarify the model. In addition, all the languages have precise semantics, so much so, that models can be fully executed, or translated into other executable formalisms, such as software code. We now briefly describe the modeling languages and their main connections. As we shall see, the general visual style, as well as many of the conventions and syntax rules, are common to all three.

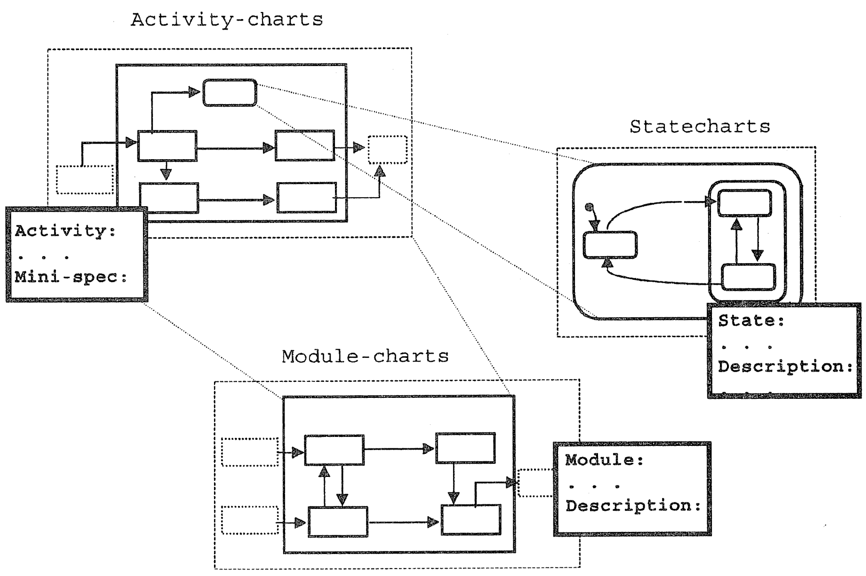


Figure 1.3 The modeling languages.

1.3.1 Activity-charts

Activity-charts can be viewed as multilevel data-flow diagrams. They capture functions, or activities, as well as data-stores, all organized into hierarchies and connected via the information that flows between them. We adopt extensions that distinguish between data and control information in the arrow types and provide several kinds of graphical connectors as well as a set of semantics for information that flows to and from nonbasic activities.

Figure 1.4 illustrates some of these notions using the EWS example. We see internal activities, such as GET_INPUT, SET_UP, and COMPARE, external activities, such as OPERATOR and SENSOR, a data-store LEGAL_RANGE, data flows, such as RANGE_LIMITS and SAMPLE, control flows, such as COMMANDS and OUT_OF_RANGE, and the control activity EWS_CONTROL, whose internal description is to be given in the language of Statecharts for the behavioral view. Notice how the hierarchy of activities is depicted graphically by encapsulation, so that a single chart can represent multiple levels of activities.

In addition to the graphical information, each element in the description has an entry in the Data Dictionary, which may contain nongraphical information about the element. For example, the activity entry contains fields called *mini-spec* and *long description*, in which it is possible to provide formal and informal textual descriptions of the activity's workings. See Fig. 1.5.

Activity-charts are described in detail in Chap. 2.

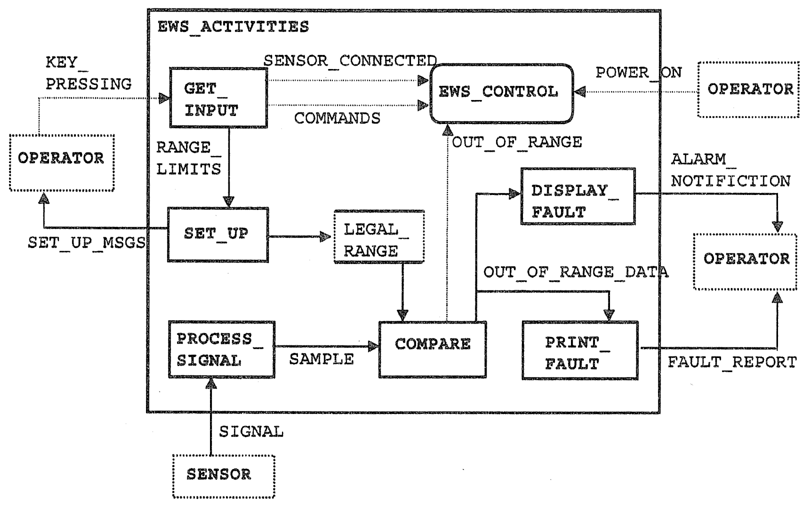


Figure 1.4 An activity-chart.

```
Activity: PROCESS_SIGNAL
Defined in Chart: EWS_ACTIVITIES

Termination Type: Reactive Controlled
Mini-spec: st/TICK;;
TICK/ $SIGNAL_VALUE:=SIGNAL;
       SAMPLE:=COMPUTE($SIGNAL_VALUE);
       sc!(TICK,SAMPLE_INTERVAL)
```

Figure 1.5 An activity entry in the Data Dictionary.

1.3.2 Statecharts

Statecharts (Harel 1987b) constitute an extensive generalization of state-transition diagrams. They allow for multilevel states decomposed in an and/or fashion, and thus support economical specification of concurrency and encapsulation. They incorporate a broadcast communication mechanism, timeout and delay operators for specifying synchronization and timing information, and a means for specifying transitions that depend on the history of the system's behavior.

Figure 1.6 contains a statechart taken from the EWS model. It consists of a top-level state EWS_CONTROL, which is decomposed into two substates. One of the substates, ON, is decomposed into two parallel behavioral components, MONITORING and PROCESSING; each of these is further decomposed into exclusive states. This means that the system must be in two states simultaneously, each from a different component. For example, when the statechart starts, the system is in WAITING_FOR_COMMAND and in DISCONNECTED. The chart also depicts events that cause transitions, such as ALARM_TIME_PASSED, which causes the system to go from the GENERATING_ALARM state to WAITING_FOR_COMMAND, and RESET, which causes the system to leave both COMPARING and GENERATING_ALARM and enter WAITING _FOR_COMMAND. Some transitions are guarded by conditions, such as the one from WAITING_FOR_COMMAND to COMPARING, which is taken when the event EXECUTE occurs but only if the condition in(CONNECTED) is true, namely, the system is in the CONNECTED state of the SAMPLING component. Some transition labels contain actions, which are to be carried out when the transitions are taken. For example, when moving from COMPARING to GENERATING_ALARM the system sends a HALT signal to the PROCESSING component.

Here, too, each element in the statechart has an entry in the Data Dictionary, which may contain additional information. For example, an event entity can be used to define a compound event by an expression involving other events and conditions. Statecharts are discussed in Chaps. 4, 5, and 6.

1.3.3 Module-charts

A module-chart can also be regarded as a certain kind of data-flow diagram or block diagram. Module-charts are used to describe the modules that constitute the implementation of the system, its division into hardware and software blocks and their inner components, and the communication between them.

Figure 1.7 shows a module-chart for the EWS. It contains internal modules, such as the control and computation unit (CCU), the SIGNAL_PROCESSOR, and the OPERATOR_MONITOR. The latter module contains the submodules KEYBOARD and SCREEN. (Here, too, the hierarchy of modules is depicted by encapsulation.) The module-chart also contains environment modules, such as OPERATOR and SENSOR, and it is noteworthy that these are similar to the external activities depicted in the functional view. The communication signals between modules include KEY_PRESSING from the OPERATOR to the KEYBOARD, the ALARM_SIGNAL from the CCU to the ALARM_SYSTEM, and so on.

Elements of the module-charts also have entries in the Data Dictionary, in which additional information can be specified.

Module-charts are described in Chap. 9.

1.3.4 Relationships between the languages

The relationships between the concepts of the three views are reflected in corresponding connections between the three modeling languages.

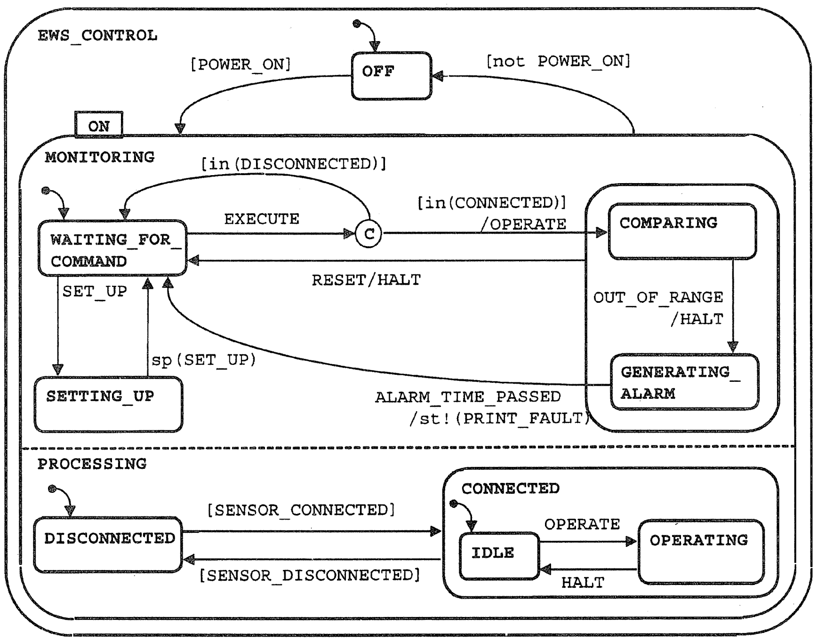


Figure 1.6 A statechart.

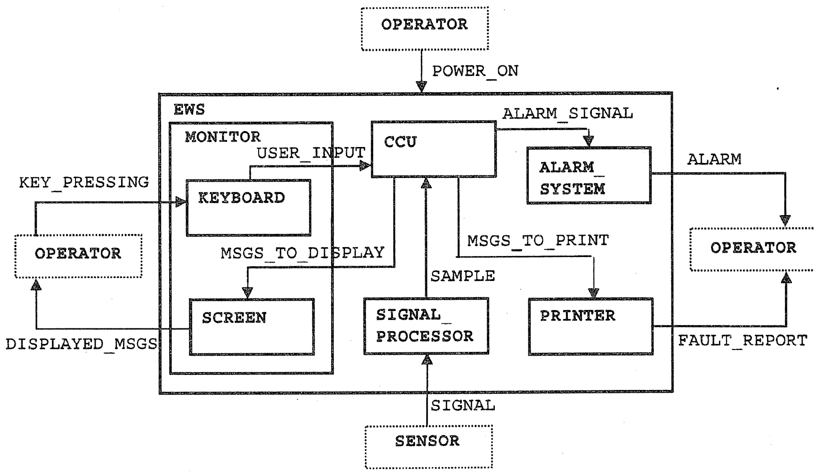


Figure 1.7 A module-chart.

Most of these connections are provided in the Data Dictionary, and they tie the pieces together, thus yielding a complete model of the system under development.

The main relationship between the functional and behavioral views is captured by the fact that statecharts describe the behavior and control of activities in an activity-chart. We thus associate a statechart with each control activity in an activity-chart. In Fig. 1.4, the @ symbol denotes that the statechart named EWS_CONTROL (which appears in Fig. 1.6) is to be taken as the "contents" of the control activity.

Another relationship between activity-charts and statecharts involves activities that are specified as being active throughout states. For example, in the Data Dictionary entry for the state COMPARING, we can specify that the activity COMPARE is active throughout (see Fig. 1.8). This means that COMPARE will start when the state COMPARING is entered and will terminate when it is exited.

There are ways to directly refer to activities from within a statechart. For example, the event sp(SET_UP), which labels a transition in Fig. 1.6, occurs when the activity SET_UP terminates (the sp stands for stopped). It causes the transition from the SETTING_UP state to WAITING_FOR_COMMAND. Chapters 7 and 8 are devoted to the connections between activity-charts and statecharts.

The relationships between the conceptual and physical models of the system are reflected in connections between activity-charts and module-charts. One such connection involves specifying which module implements a given activity. This is done in the activity entry of the Data Dictionary. For example, in the entry for the COMPARE activity we might say that COMPARE is implemented in the CCU module.

Another connection involves associating an activity-chart with a specific module in the module-chart, thus describing the module's functionality in detail. This kind of association is specified in the Data Dictionary entry for the module. For example, the activity-chart EWS_ACTIVITIES (which was shown in Fig. 1.4) describes the functionality of the EWS module. See Fig. 1.9.

Chapter 10 is devoted to describing these relationships.

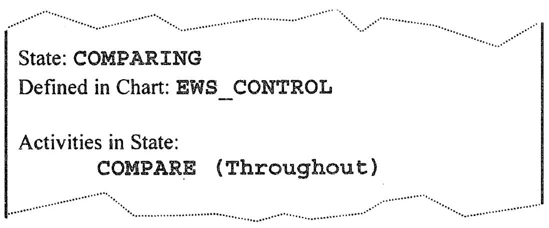
1.3.5 Handling large-scale systems

Methodological approaches, particularly the models that they recommend constructing, are essential for developing large systems. Our own approach is thus intended primarily for such systems. These involve vast quantities of information and numerous components and levels of detail, as well as portions that may appear repeatedly in many parts of the model. Such systems are usually developed by several separate teams. Our languages support features designed specifically to ease in this work.

Although a single chart can describe a multilevel hierarchy of elements, it is not advisable to overuse this capability when the model grows beyond a certain size. Accordingly, our languages allow splitting large hierarchical charts into separate ones. See Fig. 1.10, in which a separate chart is used to describe the contents of activity A.

Chapter 11 is devoted to this subject.

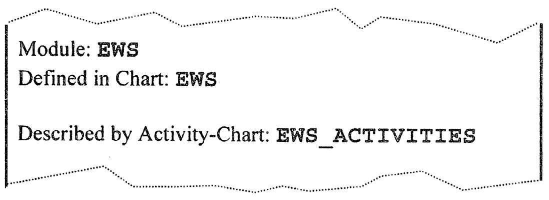
A related issue involves coping with visibility and information hiding by setting scoping rules of elements in the model. It is also possible to introduce global shared information in a model component called a



State: **COMPARING**
Defined in Chart: **EWS_CONTROL**

Activities in State:
    COMPARE (Throughout)

Figure 1.8 Specifying an activity throughout a state.



Module: **EWS**
Defined in Chart: **EWS**

Described by Activity-Chart: **EWS_ACTIVITIES**

Figure 1.9 An activity-chart describing a module.

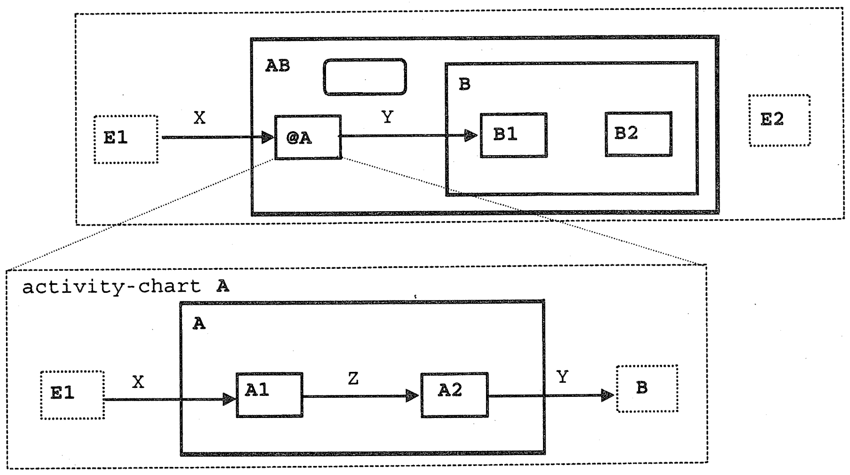


Figure 1.10 Splitting up charts.

global definition set. This is analogous to the scoping issue in programming languages.

Scoping is discussed in Chap. 13.

A very important feature of our languages is that of *generic charts,* which allow reusing parts of the specification. A generic chart makes it possible to represent common portions of the model as a single chart that can be instantiated in many places, and in this it is similar to a procedure in a conventional programming language.

Generic charts are described in Chap. 14.

Another feature that contributes to reusability is that of *user-defined types,* which are described in Chap. 3. This feature makes it possible to define a data type that will be used for many data elements in the model.

1.4 The STATEMATE Toolset

We now provide a very brief description of the STATEMATE toolset (Harel et al. 1990), which supports the languages and approach presented here. STATEMATE was intended primarily to help address the goals of the specification stage, although it supports some of the activities carried out in other stages, too. See Fig. 1.11 for a schematic overview of the STATEMATE toolset.

We should note that the modeling approach and languages presented here have a life of their own, whether they are used in conjunction with a computerized tool or not. Moreover, there are other tools, both commercial and of research nature, that support Statecharts and other aspects of the approach. We describe STATEMATE here both because we have been part of the team that designed it and because it still seems to be the most powerful tool of its kind available.

For entering the information contained in the model, STATEMATE has graphic editors for the three graphical languages, as well as a Data Dictionary. It carries out syntax checking and tests for consistency and completeness of the various parts of the model. While constructing the model, the specifier can link original textual requirements to elements of the model. These links can be used later in requirement traceability reports. STATEMATE also provides extensive means for querying the model's repository and retrieving information from it. A number of fixed-format reports can be requested, and there are document generation facilities with which users can tailor their own documents from the information constituting the model.

Our view of system development emphasizes "good" modeling, but it also regards as absolutely crucial the need to enable a user to run, debug, and analyze the resulting models and to translate them into working code for software and/or hardware. Accordingly, STATEMATE has been constructed to "understand" the model and its dynamics. The user can then execute the specification by emulating the environment of the system under development and letting the model make dynamic progress in response.

Using STATEMATE the model can be executed in a step-by-step interactive fashion or by batch execution. In both cases, the currently active states and activities are highlighted with special coloring, resulting in an (often quite appealing) animation of the diagrams. It is also possible to execute the model under random conditions and in both typical

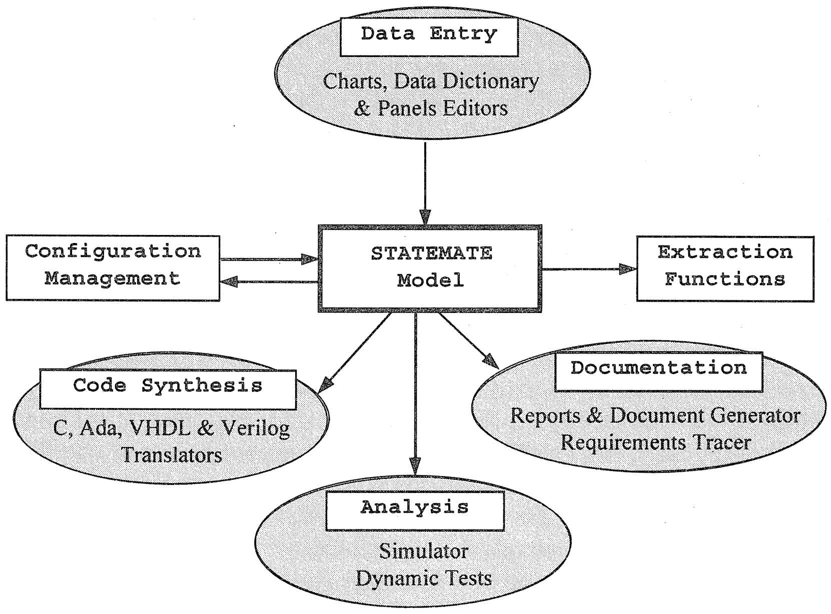


Figure 1.11 The STATEMATE toolset.

and less typical situations. A variety of possible results of the executions can be accumulated to be inspected and analyzed later.

We should note that it is possible to execute only part of the model (in any of the execution modes), as long as the portion executed is syntactically intact. This implies that there is no need to wait until the entire model is specified to carry out executions, and even an incomplete model can be executed and analyzed. Moreover, it is possible to attach external code to the model, to complete unspecified processing portions, to produce input stimuli, or to process execution results on-line. This openness enables STATEMATE to be linked to other tools.

STATEMATE also supports several dynamic tests, which are intended to detect crucial dynamic properties, such as whether a particular situation can be reached starting in a given state. These tests are carried out by the tool using a form of exhaustive execution of scenarios. We shall not get into a discussion of the feasibility of such exhaustive executions here; the reader is referred to Harel (1992b) for some comments on the matter.

Once a model has been constructed, and has been executed and analyzed to the satisfaction of the specifiers, STATEMATE can be instructed to translate it automatically into code in a high-level programming language. This is analogous to the compilation of a conventional program into assembly language, whereas model execution is analogous to its direct interpretation. Currently, translations into Ada and C are supported, and a variant of STATEMATE enables translation into hardware description languages VHDL and Verilog. Code supplied by the user for bottom-level basic activities can be appropriately linked to the generated code, resulting in a complete running version of the system. The resulting code is sometimes termed *prototype code,* because it is generated automatically and reflects only those design decisions made in the process of preparing the conceptual model. It may not always be as efficient as final code, although it runs much faster than the executions of the model itself, just as compiled code runs faster than interpreted code. For some kinds of systems, however, this code is quite satisfactory.

One of the main uses of the synthesized code is in observing the model performing in circumstances that are close to its final environment. The code can be ported and executed in the actual target environment, or as is more realistic in most cases, in a simulated version of the target environment. To this end, STATEMATE makes it possible to construct a "soft" version of the user interface of the final system, which can then be activated, driven by the synthesized code. The resulting setup can be used to debug the model by subcontractors and customers, for example. Again, Chap. 16 contains a discussion of how such code can be beneficial in the design phase of system development.

Associated with the code synthesis facility is a debugging mechanism with which the user can trace the executing parts of the code back up to the model using back animation. The requirements traceability feature makes it possible to trace problems back up to the (textual) requirements.

For more on these topics, we refer the reader to the STATEMATE documentation supplied by I-Logix, Inc.