Chapter

16

Transition to Design

The main thrust of this book has been the careful description of a set of languages for modeling complex reactive systems. A model built with these languages can be used during the system development process in various ways, depending on the development method adopted. We shall call such a model the STATEMATE model, and as explained briefly in Chap. 1, it is commonly used in the specification phase.

Now that we have finished describing and illustrating the modeling languages themselves, we would like to come full circle and return to the issue of positioning the resulting specification models within the entire development process. In most development schemes, the design phase comes after the specification and leads to the implementation. Because different implementation technologies give rise to different concerns and different criteria for the quality of the design, the methods used in the transition from specification to design will differ, too.

This chapter concentrates on using the STATEMATE model for specification and on the main ways for carrying out the transition from the specification model to the design.

16.1 STATEMATE Models in the Development Process

There have been a number of proposals for defining the life-cycle process of system development. Among these are the classical waterfall model (Boehm 1976; Royce 1970), the rapid throwaway prototyping approach (Gomma and Scott 1981), the evolutionary prototyping (Gomma 1986; McCracken and Jackson 1986), and the spiral model (Boehm 1988). In general, such development processes start with requirements analysis, during which the system's specification is constructed. Although specifications—and therefore the models describing them—are treated differently in the various approaches, the

subsequent phase is almost always design, which is an essential prerequisite to implementation.

A STATEMATE model can be used in the specification phase of most of the development processes. Nevertheless, it is important to understand the model's role in the overall process and how the information contained in it can be used in subsequent phases, mainly the design phase. We now show how STATEMATE models can be used in various ways during specification.

16.1.1 Models as prototypes

One common approach in developing complex systems is to use the specification model for prototyping. Because a STATEMATE model is executable, it lends itself nicely to this purpose. In fact, executable specifications—termed *operational* by Zave (1984)—are the basis of the "rapid throwaway prototyping" approach of Gomma and Scott (1981). The structure of prototyping models in this case is problem-oriented, not implementation-oriented, and the system's external behavior is studied by executing the model. The observed behavior must then be preserved during transition to design, a topic we take up in Sec. 16.2.2.

When the system is to interface with humans, developers can use special tools to build images of screen windows in computer systems or mock-ups of the control panels that will eventually make up the actual interface. If and when the interface of the model with its environment (i.e., the input/output data) is defined as it will be in the implementation, this information can be transferred to the design phase. Other portions of the model are not directly used in the design, and most of the design has to be started from scratch.

16.1.2 Design using specification models

The second approach calls for using the specification model directly to obtain a design. In other words, STATEMATE modeling is used as a high-level implementation language. Because the model can then be translated automatically into the target implementation by code synthesis, this process can be viewed as true compilation.

In this approach, the principal structure of the implementation is determined in the specification phase. This is one of its disadvantages, because the specifier must consider issues that are not relevant to this phase. However, the main advantage is continuity, which is hailed as one of the main virtues of object-oriented development methods. If an object-oriented approach is indeed taken, objects from the problem domain are refined and appear in their new guise in the implementation structure. Work that was carried out in the first phase is not lost. There are better means for traceability, and the resulting systems are therefore easier to maintain. The transition to

design is thus more reliable, depending mainly on the reliability of the code-synthesizer.

16.1.3 Restructuring for design

A "middle of the road" approach (i.e., not exactly full prototyping and not exactly designing during the specification, but a little bit of both) is restructuring the specification for design. There are two issues that raise the need for a new model for design purposes, and hence for such restructuring to take place:

- Specification is carried out in the problem domain, and design in the implementation domain.
- There are many different kinds of implementation frameworks, which requires using different design languages.

Restructuring requires the system developer to allocate elements of the model to elements of the design. Thus the STATEMATE specification model is not discarded. Rather, new constructs are prepared for the system's design, and the constructs of the STATEMATE languages are mapped into them.

We should remark that although design is often carried out in a language external to the STATEMATE framework, some parts of the design can be carried out within STATEMATE, using module-charts and their connections with other parts of the STATEMATE model. See Chaps. 9 and 10.

16.2 Mapping Models to Design Structures

We now discuss "real" transition to design, that is, restructuring the specification model and mapping it into the design structures, as introduced in Sec. 16.1.3.

Specification structures must be mapped into a particular configuration of implementation resources. Ideally, we would like automatic transformations that preserve external behavior but change the mechanisms that produce that behavior. This is usually not the case, so we will more commonly use heuristics that recognize structures in the specification model and transfer them to the design model. In general, this transition to design involves a number of concepts regarding the usage of design criteria for obtaining a mapping and the evaluation of the resulting mapping. Some of these are common to the different target environments, and some are more specific to particular implementation technologies.

16.2.1 Design criteria

Different architectures are used for different target implementation technologies, and each case involves different design considerations.

Moreover, the level of design can vary, too: one could decide to work on a high level of system design, in which there is a division into subsystems, or on lower levels, in which there is a mapping into the actual constructs of the final software or hardware.

In high-level design, especially in real-time embedded systems, there will often be both software and hardware components. In such cases, the first stage in the mapping to design involves allocating the functional requirements described by the system-level specification model to software and hardware components. This is done by the systems engineer, according to a variety of criteria: the basic nature of the function (e.g., certain things can only be done with hardware, such as sensing information from the physical world), desired performance, existing components, etc.

The decomposition itself can be carried out in STATEMATE mapping from activity-charts to module-charts (using the "implemented by module" relationship), as discussed in Chap. 10. One such case, used in MIL-STD-498, was described in Chap. 15, that is, the division of a system into its software and hardware components (the CSCI and HWCI of the standard, respectively).

When we carry out this design decomposition, some additional issues have to be dealt with these concern requirements management, deriving interfaces from the allocation, and traceability concerns. Many of these are discussed in articles appearing in the two first chapters of Thayer and Dorfman (1990), and some of them can also be carried out within the STATEMATE languages.

If a system is pure software (or if we are in later levels of software/hardware design), we will reach the need to map into software components. This, too, depends on the target technology. There are essentially three general issues here. One is the fact that since we have an orderly, complete and consistent specification model, we are in a position to identify patterns of similarity in the functional components. This makes it possible to make decisions regarding the mapping of, say, similar functions that appear in different processes of the behavioral specification into a single software function with parameters or to a class with several instances or subclasses.

The second issue is that of an object-oriented target implementation, an extremely popular and beneficial approach in recent years. While objects are very useful in the implementation stage, some systems are better thought of at the early stages of development in nonobject ways. In such cases, the issue is to transfer requirements based on functional decomposition into an object-oriented implementation. This is discussed in several places; see, for example, Ward (1989) and Gomma (1993). Some of the methods are based on Ward/Mellor or Hatley/Pirbhai specifications, but they hold also for function-based STATEMATE models. Other possibilities involve generating scenarios (or the more

general use-cases) from the STATEMATE model and to proceed from there as in object-oriented design. Of course, if the modeling itself is carried out—already in the specification stage—in an object-oriented fashion, the mapping to an object-oriented design will be straightforward. The generic-charts construct of our languages (see Chap. 14) can help with this because a generic chart is a natural candidate to be a class in the implementation.

The third issue is that of specific kinds of applications, such as hard, real-time systems. For these, there are usually more singular criteria and specialized considerations of performance. Some techniques and components available in specific implementation environments are concurrent tasks, synchronization and communication mechanisms, and timers of the particular real-time kernel. Good coverage of the design process for real-time systems, starting from the specification is given by Gomma (1993).

In principle, many of the special criteria can be embodied in algorithms for carrying out the mapping. The automatic code generation can be based on them, with some user guidance (employing various compilation profiles) about such questions as to whether an activity should be translated into a function or a task, whether to use a polling method for some particular input or interrupt, and so on.

This guided translation into code has proved itself very well in the arena of pure hardware (e.g., ASIC). Chip designers use VHDL and Verilog, high-level design languages similar to programming languages, to express their designs (Smith 1996). (This code can be later automatically transformed into chip schemes by commercial tools.) The designers can write the code manually; alternatively, they can develop STATEMATE models, which are translated into VHDL and Verilog by automated tools (such as the translator developed by I-Logix). There are also compilation profiles to guide the translation. These profiles are based on the designer's knowledge and the criteria he or she applies in the particular case at hand. The profiles contain high-level instructions, such as putting several model components in the same design entity and defining the port signals, and low-level decisions, such as how the signals will be implemented (e.g., the polarity of conditions).

16.2.2 Evaluation of the mapping

Once a mapping from the model to the design has been constructed, it should be checked for completeness and consistency, from both structural and behavioral points of view. The formality of our languages obviously make such tests possible in principle, and indeed the STATEMATE tool supports a broad variety of them.

First, we have to make sure that all the requirements have been covered, for example, that all the functions in the specification model have

been mapped to structures in the design. Conversely, we must show that all parts of the design have some source in the specification model. The better and more detailed these links are between the specification model and the design, the easier it is to carry out forward and backward traceability, which is crucial for convenient maintenance of the system under development.

We also have to check the structural consistency of the mapping. This includes several things, such as consistency of the hierarchy and of the interfaces. For example, if an activity `A` was mapped to some design structure `M`, and a subactivity `B` of `A` was mapped to `N`, then `N` must be a substructure of `M` in an appropriate sense. Similarly, if a data-item `X` was specified as flowing from an activity `A` to an activity `B`, there must be a way for `X` (or its mapped image) to flow from the design structure implementing `A` to that implementing `B`.

The behavioral aspects are far more problematic. We mentioned that behavior must be preserved by the mapping. Put simply, we want things we know about the behavior of the model (e.g., those discovered by executing it or by running the synthesized code) to hold for the implementation, too. Of course, we could run the implemented system and check that the scenarios match. However, it is necessary to emphasize that running or executing models and designs usually cannot guarantee full behavioral consistency because the number of possible scenarios will often be infinite or at best unreasonably large. As Dijkstra once put it, testing and debugging cannot be used to demonstrate the absence of errors, only their presence.

What is needed for air-tight confidence in the system's desired behavior is true program or system *verification.* Because verification is a whole science in itself, we shall not dwell on it here, except to make a few general comments. Good basic books on verification include Francez (1991) and Loeckx and Seiber (1984).

When we use the term *verification,* we mean rigorous mathematical proofs of correctness. In our framework, this means proving that the mapping indeed preserves behavior under all circumstances. Even this needs to be more carefully stated. For example, we might want to know that the values of certain variables are preserved in the transition to design or that certain kinds of sequences of events that take place in executing the model will also take place in the implementation.

The three basic facts about verification in our framework are as follows (see Chap. 5 of Harel, 1992*a*):

- The general verification problem is noncomputable. This means that we cannot hope for a verification tool that will be able to routinely prove the correctness of any mapping.

- In principle, a correct mapping can be proved correct in an appropriate mathematical setup, so that this direction of work is definitely worth pursuing.
- In recent years there have been major advances in techniques and automated tools for verifying systems. Hardware industries are beginning to use them on real systems, and the feeling is that software and embedded systems will not be long in following this lead. These modern verification methods are based on specifying properties in temporal logic and on model-checking techniques. A large amount of material can be found in Manna and Pnueli (1992).