Chapter 3 (JZ): Structured Approach to Real-Time Software Design

The chapter shall present principles of a structured approach to real-time software development. A typical software architecture of a real-time system will be discussed, including all major components: external process interface, user interface, communication link, mass storage handler, processing component and timer. Characteristics to real-time systems design patterns will be discussed, including those specific to concurrency and distribution.

3.1. Introduction
3.2. Principles of Real-Time Structured Analysis and Design
3.3. Cruise Control Problem
3.4. DARTS: Design Approach to Real-Time Systems
3.5. Cruise Control Problem Revisited
3.6. Case Study: Boiler Control System
3.8. References

3.1. Introduction

In general, the design of real-time software starts by converting a requirements specification into an architectural model of the system. This is, in principle, done beginning with a context diagram, and is true for both approaches, whether structured or object-oriented. What happens next, that is, how the information in a context diagram and the specification requirements are converted into the design, is a matter of using a specific development methodology. Of the three major components of every software design methodology, method, techniques, and tools, we focus in this chapter on the method, that is, the design notation, and the techniques, that is, steps to develop the software design.

The simplest approach to software design, which we could call the infancy approach, is (1) to define software structure by dividing it into modules called from the Main module, and (2) define the behavior of each module in a flowchart. The diagram defining software structure that way is called a structure chart and is shown in Fig. 3-1 for a weather monitoring system, which reads certain physical quantities from respective sensors and presents the readouts on the display. Example of a flowchart for a sample module to read temperature sensors is shown in Fig. 3-2.

These types of diagrams, although suitable for representing designs of simple programs, do not have sufficient expressive power to represent characteristics of more involved software designs. To adequately describe the structure and behavior of real-time software, one needs some more sophisticated notation, focusing on various aspects of interacting with the environment. Such notation has been in use since mid-1980s and is called Real-Time Structured Analysis and Design Technique (SADT-RT).

3.2 Principles of Real-Time Structured Analysis and Design

Historically, structured notation for real-time systems has its origin in the work of Mellor and Ward [1], refined by Hatley and Pirbhai [2]. The description of the structure and behavior of software under development is presented using two principal types of diagrams: data flow diagrams (DFD) and control flow diagrams (CFD), which in principle can be combined. The notational elements of both are presented in Fig. 3-3.

Two major components of a DFD are functions (or processes) representing data transformations (shown graphically by circles) and data flows representing relationships between processes via data inputs and data outputs (shown graphically by directed arcs). A distinction between discrete and continuous data flows is marked by a double arrow in
case of continuous data. Data sources and data sinks are known as data stores and shown graphically by two parallel bars.

Since the DFDs only show what functions the system performs, without taking into account the circumstances under which these operations are executed, CFDs were invented to facilitate defining the conditions of operation. DFDs represent data transformations, while CFDs represent the transformations in the mode of operation. In this respect, CFDs are somehow similar to state transition diagrams, but in this notation state transition diagrams play a slightly different role, anyhow. The notation for CFDs differs from that for DFDs in circles and arcs, which are both drawn with dotted lines, not solid ones as in DFDs. In modern terminology, CFDs represent events, that is, data items, which are short-lived or transient, but in principle they are not much different from DFDs, which represent permanent data transformations. Three essential kinds of events are distinguished as represented by CFDs: trigger, enable, and disable, and marked by labels at the control arcs, with capital letters T, E, and D, respectively.

From the real-time design perspective, it is important to note that in SADT-RT only continuous data (marked with double arrows) and control processes (CFDs) can retain any timing information, that is, extend their behavior over time. All other processes are instantaneous, without an internal state. To represent this kind of information, data processes which use continuous data are defined with so called minispecs. Control processes are described in terms of finite state machines represented by state transition diagrams. There is also one significant structural difference between DFDs and CFDs: CFDs may contain a single bar representing an interface with their specification, which normally is a state transition diagram. All inputs and outputs of the control specification (state transition diagram, in this case) must come through this bar symbol. Example of using this feature is provided in the cruise control case study.

In addition to DFDs, CFDs, state transition diagrams and minispecs, there are other notational vehicles to provide more detailed design information. Among those, there are entity-relationship diagrams and structure charts. These, although important, are less specific to defining real-time behavior of software, so they will not be considered here. One must also note that all textual information about the design is kept in the Data Dictionary containing an alphabetical listing of all the data and control flows on the DFDs and CFDs with their definitions.

The major technique to develop software design from such structured notations is hierarchical decomposition, also known as leveling. Graphical models of software designs are built hierarchically, beginning from the context diagram, known as Level 0. Each lower level diagram provides a more detailed description of the particular process. This is illustrated in Fig. 3-4. When a particular process is not decomposable further, forming a leaf in the decomposition hierarchy tree, it is called primitive. Then its function must be described using some kind of precise notation, such as structured English, pseudocode, flowchart, etc., presenting details of an algorithm or some other computation performed by this primitive process.

Once a complete decomposition is done, for both DFDs and CFDs, a system architecture is developed. The system architecture is understood here as a statement of system’s physical configuration in terms of the architecture modules, their interconnects and the information flow between them. In practice, the system architecture diagrams are not much different from DFDs and CFDs. The architecture embraces all the information included in DFDs and CFDs, only organizing it in a new way according to the architectural principles.

The most representative book describing this technique [2], advises to start designing software with the context diagram and then proceed according to the following steps:

1) Perform data flow decomposition of system’s functions into processes and interfaces among them.
2) Perform control flow decompositions into processes and interfaces among them.
3) Develop process specifications (minispecs) in structured English or equivalent technique.
4) Develop control specifications using state transition diagrams or equivalent technique.
5) Develop response time specifications, which define the time limits on events in the system.
6) Develop data dictionary.
7) Produce the architecture expressed in terms of flow diagrams (allocating the functional processes of the model to physical units of the system) and interconnect diagrams, and their specifications, including data dictionary.

The key element in design with this method is the architecture model derived from DFDs and CFDs using the template of the architecture. It relies on the principal characteristics of the system as captured in DFDs and CFDs, but refines
those diagrams into a physical representation. Such template is composed of four basic components depicted as rectangular boxes tied together (Fig. 3-5):
- functional and control processing (central box)
- input processing (situated on the left of the central box)
- output processing (situated on the right of the central box)
- user interface (situated on the top of the central box).

The fundamental idea in creating the architecture in SADT-RT according to [2] is to develop so called enhanced requirements model, which means, in fact, mapping all the DFDs and CFDs onto the architecture template, by placing them in respective boxes outlined in Fig. 3.5. An example of an architecture template for aircraft’s flight management system is presented in Fig. 3-6. The actual boxes are filled with details after the DFD/CFD analysis has been done. For a description of those details the reader is referred to [2].

To illustrate the use of this method, we are presenting in the next section the case study of an automobile cruise control system following [2].

3.3. Cruise Control Problem

To illustrate their design method, Hatley and Pirbhai [2] give an example of developing a Cruise Control System (CCS). The academic formulation of this problem has its origins in one of the DOD conferences in mid-eighties, when multiple teams were asked to present their designs using different methods, for comparison. The excerpt of the problem statement taken from [2] is presented below. Converting this problem statement into the actual Software Requirements Specification is a good topic for an exercise, and is subject of Exercise 3-1, so we will pursue here directly with the design.

**Cruise Control Problem Statement [2]**

*Note.* This cruise control problem is only a part of a larger Automobile Management System, as described in [2]. Other parts of this system include average speed monitoring, fuel consumption monitoring, and maintenance monitoring. In addition to that, we are not concerned in this book with the calibration capability of the cruise control, so its description is marked below by brackets.

The cruise control function is to take over the task of maintaining a constant speed when commanded to do so by the driver. The driver must be able to enter several commands, including Activate, Deactivate, Start Accelerating, Stop Accelerating, and Resume. The cruise control function can be operated any time the engine is running and the transmission is in top gear. When the driver presses Activate, the system selects the current speed, but only if it is at least 30 miles per hour, and holds the car at that speed. Deactivate returns control to the driver regardless of any other commands. Start Accelerating causes the system to accelerate the car at a comfortable rate until Stop Accelerating occurs, when the system holds the car at this new speed. Resume causes the system to return the car to the speed selected prior to braking or gear shifting.

The driver must be able to increase the speed at any time by depressing the accelerator pedal, or reduce the speed by depressing the brake pedal. Thus, the driver may go faster than the cruise control setting simply by depressing the accelerator pedal far enough. When the pedal is released, the system will regain control. Any time the brake pedal is depressed, or the transmission shift out of top gear, the system must go inactive. Following this, when the brake pedal is released, the transmission is back in top gear, and Resume is pressed, the system returns the car to the previously selected speed. However, if Deactivate has occurred in the intervening time, Resume does nothing.

[Since speed and distance per drive shaft rotation are affected by tire size and wear, the system is to have a calibrate capability. For this purpose, two further controls will be provided: Start measured mile and Stop Measured Mile. They are only effective when cruise control is inactive. The driver presses Start Measured Mile at the start of measured]
mile, drives the mile, and then presses Stop Measured Mile. The system is then to record the number of shaft pulses over this interval, and use this count as its mile reference in all calculations.]

The context diagram for this problem is shown in Fig. 3-7. The major task assigned to the CCS software is to maintain constant speed by controlling the setting of the throttle, based on several inputs, such as gear position (top gear or not), engine status (ON or OFF), break pedal (pressed or not), current speed (within preset limits), and cruise buttons (cruising activated or not, acceleration started or stopped, and cruising resumption). On the context diagram, we do not follow exactly the rules of SADT-RT. We produce a combined context diagram for both DFDs and CFDs, because it is simpler without a loss in generality (separate diagrams for DFDs and CFDs represent the same process, anyway). We do not distinguish, either, the graphical differences between continuous and discrete signals (that is, control arcs are not marked with dotted lines). It must be also noted that we deal only with the cruise control part of the entire automobile management system in the original problem description from [2], so our diagrams will be simpler but illustrative enough to show the essence of the SADT-RT method in Hatley/Pirbhai version.

First, the top-level Cruise Controller process is split into lower level functions, according to steps 1 and 2 from previous section (remember that we combine DFD and CFD information into a single diagram). These functions, shown in Fig. 3-8 as circles representing respective processes (data transformations), are as follows:
- Select Speed,
- Maintain Speed,
- Maintain Acceleration.
They form the first-level internals of the Cruise Controller. Once this happened, we need to define each of those processes in detail. Normally, there are two options in SADT-RT: either split each process further down into its internal functions, according to hierarchical organization, as shown in Fig. 3-4, which would be a repetition of steps 1 and 2, or, if the function of the process is simple enough, just define its transformation using some rigorous notation to express the algorithm, which means applying step 3. All three internal processes of the Cruise Controller are defined in the latter way, as shown in Fig. 3-9.

An important consideration in the first-level decomposition of the Cruise Controller is given to the four transient signals being a part of the CFD, since they do not convey data in a classical sense (Fig. 3.8): Gear Position, Brake Pressed, Engine Running and Driver Command. In this design, they are not delivered to any lower-level CFD, because it is not decomposed further, so they have to end on a bar symbol inside the CFD. Furthermore, since the CFD of the Cruise Controller is not split into lower level CFDs (only DFD is), it must be defined in terms of a state transition diagram or equivalent (step 4). A respective state transition diagram is shown in Fig. 3-10. To keep the diagram reasonably clear and not overloaded, it does not include transitions corresponding to Activate and Deactivate commands (see Problem Statement).

Steps 5 and 6 of the SADT-RT are self-explanatory, so we do not repeat them here. However, step 7 is crucial in the design, since it produces the final architecture. As mentioned in previous section, the architecture template, similar to that shown in Fig. 3-5 is superimposed on the context diagram to extract architectural elements of respective categories: functional processing, input, output, and user interface. The result of combining that way diagrams from Fig. 3-5 and Fig. 3-8 is shown in Fig. 3-11. This is the architecture of the Cruise Controller. It contains the following software components:
- Functional Unit, the heart of the Cruise Controller, which performs the computational functions and interacts with three other components
- User Interface, which supplies driver commands to the Functional Unit
- Input Processing, probably the most comprehensive unit, since it receives all sensor data, and
- Output Processing, which emits the throttle setting signal.
Technically, this diagram is developed further in [2] into architecture flow and interconnect diagrams, but the resulting division into software components does not differ from that presented in Fig. 3-11.

3.4 DARTS: Design Approach to Real-Time Systems

A more refined approach, derived from Ward/Mellor and Hatley/Pirbhai structured design, presented by Gomaa [3], is called DARTS (abbreviated from Design Approach to Real-Time Systems). Its contribution relies on explicit inclusion
of concurrency into the design, as well as on expressing communication and synchronization mechanisms among units of concurrency. It evolved later into ADARTS (Ada DARTS) and CODARTS (Concurrent DARTS), but its essential characteristics did not change.

The basic notation in DARTS is the same as in SADT-RT, but due to inclusion of tasking and interprocess communication several notational vehicles have been added, as shown in Fig. 3-12. They include building blocks for tasks themselves and modules for information hiding, as well as arcs ending with arrows for events and three kinds of message queues: loosely coupled messages and tightly coupled messages with and without reply.

DARTS does not diverge at all from SADT-RT in developing DFDs and CFDs, starting the entire procedure from the context diagram. In other words, all steps 1-6 from SADT-RT as stated in Section 3.2 above, apply to DARTS. The only significant difference is concerning the development of tasks, which starts after all DFDs/CFDs and their hierarchical specifications have been completed. In summary, the steps in DARTS leading to the development of a software design are as follows:

1) Develop structured system specification according to SADT-RT, as described in steps 1-6 above.
2) Structure the system into concurrent tasks, developing multiple tasks (whenever applicable) for each leaf node of the DFD/CFD hierarchy, which is the result of the previous step.
3) Define task interfaces, by mapping data flows between tasks to message queues, event flows to signals, and data stores to information hiding modules. Timing analysis can be done at this step as well, since all program components (tasks) and their relationships are known.
4) Design each task, which is, in principle, a sequential program potentially exchanging information with other tasks (sequential programs).

To illustrate the use of DARTS, we will analyze again the development of a Cruise Control System, as described in [3]. As in the case of SADT-RT application to cruise control, described in previous section, we are only concerned with the part of the solution dealing directly with Cruise Controller, and will be ignoring the rest of it, which is related to the entire automobile management system.

The development starts with the context diagram, almost identical to that presented in Fig. 3-7. There are only two insignificant differences. First, the context diagram in [3] does not include the gear sensor, and, second, it does include an interface to timer. As far as gear sensor is concerned, its lack or existence in the context diagram does not change anything in the resulting software design, since it is just another sensor value read by a specific task dealing with data acquisition. Regarding the timer, we will assume that time is taken internally from the processor’s clock, which is a reasonable assumption, since explicit dealing with timer location is only critical in distributed systems, where timers located on different computers have to be synchronized. A cruise control system may be implemented on multiple processors, but due to their mutual proximity it will not be considered a distributed system, unless interprocessor communication takes a significant time compared to computations. In addition to that, as in the previous example, we are not dealing with calibration, to simplify the example.

With these assumptions in mind, we can proceed with the design according to the four steps listed above. First, beginning with step 1, that is developing respective diagrams of SADT-RT, it must be noticed that the structure of the design given in [3] is slightly different from that presented in [2], even though the cruise control problem is essentially the same. The main difference is that in [3] each input function (sensors, command interpretation) is represented by a separate entity (circle) in the DFD, which represents a decomposed context diagram, while in [2] all input functions are lumped into a single CFD, represented by a bar symbol shown in Fig. 3-8. Both decompositions are equivalent, however, as long as the sensor data remain the same.

Nevertheless, one significant difference between the two designs should be noticed, namely, that in state transition diagrams. The number of states in [3] has been significantly increased, from three to eight, including, in addition to principal states listed in Fig. 3-10, Idle, Accelerating and Cruising, five new states as follows: Initial/Braking, Initial/NotBraking, CruisingOff/Braking, CruisingOff/NotBraking, and Resuming. The decision whether it is essential to have a larger number of states represented in the design or not, should be based on the level of accuracy required for the proper quality of the design, as well as on the resulting complexity. For our purposes, it is sufficient to deal with three states, since our objective is only to understand the design method. In principle, by the very nature of state transition diagrams some states of the 8-state transition diagram can be combined to form a 3-state diagram.
Based on the details of decomposition of the context diagram, presented in previous two paragraphs, we can now structure the system into concurrent tasks, according to the rule of step 2 in DARTS. First, regarding inputs, all sensor readouts (brake, engine, gear) can be essentially combined into a single task. Next, reading driver commands is very different from sensor readings and should be handled by another task. Finally, measuring the speed and acceleration is also significantly different from other input processing, so this function should be allocated to a separate, third task.

The only output function, setting the throttle position, is naturally assigned to the fourth task, and the central function of doing the computations is assigned to task number five. The result of this structuring is shown if Fig. 3-13. This diagram also includes descriptions of interfaces between respective tasks, as required by step 3 of DARTS. It is interesting to note that the original solution in [3] includes a sixth task, similar to MaintainSpeed data transformation unit from Fig. 3-11, but this is really unnecessary, since this function can be included in the CruiseControl task in Fig. 3-13. At this point, having all tasks and their interfaces defined, one has to determine each task’s structure, according to step 4. A sample structure of the central task to perform computations is shown in Fig. 3-14, using the structure chart notation.

### 3.5. Cruise Control Problem Revisited

Both solutions outlined above, SADT-RT and DARTS, have one major problem, so do many other solutions in the software engineering literature. They do not specify a critical part of both designs, the control algorithm, which may have serious consequences to the system performance. For example, let the Desired Speed be 60 mph and Current Speed 50 mph. Assume that the Resume button is pressed. How fast would it take for a car to reach the Desired Speed? Would it be a few seconds, a minute, or longer? What will be the speed trajectory to reach the Desired Speed, that is, will the speed slowly increase from Current Speed to reach the Desired Speed, or will the value of speed increase above the Desired Speed value and then drop to it? Or maybe the speed will oscillate around the Desired Speed, taking values slightly above it and slightly below it, interchangeably? Fig. 3-15 illustrates some of these possibilities.

In this respect, Hatley/Pirbhai solution is a bit more specific, because it includes, in fact, the proportional control algorithm, without saying it explicitly. The formula to maintain speed, in Fig. 3-9, defines the value of the control signal in relation to the deviation of the controlled variable (speed) from the desired value (desired speed). The relationship is proportional. Control engineering knows, however, more optimal control laws and that has to be considered when a software controller is designed.

Proportional control is the simplest possible feedback control algorithm and involves multiplication of an error (in our example, the difference between current speed and desired speed) by a constant coefficient, called control gain, that amplifies the error signal. This results in applying the stronger control value the higher the error is. In case of a CCS, the throttle setting, that is, the voltage applied to opening the throttle is being affected accordingly. Since every physical system is subjected to energy limitations, we have to restrict the rate of changing the throttle settings. Therefore there is an additional inequality, which must be met by the derivative of the throttle setting, that is, its rate of change.

It is characteristic to the proportional control systems, that for certain types of plants the output of the control system has a steady-state error (offset), for example, the actual speed differs by a small value from the desired speed in a steady state (Fig. 3-15). A common sense explanation is as follows [4]. If the Current Speed equals the Desired Speed, then the difference (error) is zero, so the proportional controller with shut the throttle and the car will start slowing down. This situation will continue until the system will find an equilibrium state, where the actual speed is enough lower than the Desired Speed, so the throttle displacement is sufficient to maintain the speed close the Desired.

This error can be eliminated by adding another component to the control equation, namely, an integral, which makes the output of the controller a function not only of the current value of error but also of the previous values. Thus, the control action is generated using two terms, proportional control (P) taking into account current values of error, and integral control (I) taking into account error history. This is known as PI control. The interpretation of the integral controller is to take control action even if the error is very small but of the same sign over a long period of time, which means the existence of an offset. Naively thinking, the speed, which is a derivative of distance, needs to be integrated to change the throttle opening as displacement (which itself is a distance).

6
In proportional control, if the error has different signs over a period of time, it can mean that there are oscillations of the actual value of the controlled variable. In such case, adding an integral control, which is based on history, will not help, and one has to apply predictions. Technically, one can take into consideration the derivative of the error signal and include this term into the control equation. This technique is called derivative control (D) and the controller including all three terms – the PID controller. Derivative control is used to provide anticipative action by making a linear prediction of error (Fig. 3-16). For processes with significant delays, anticipating behavior is not very effective, so in case of a cruise controller, normally PD control is sufficient. These considerations give a reason why real-time software engineers need understand basics of control theory or, at least, interface well with control engineers. More information on PID controllers can be found in elementary textbooks on control engineering [5,6].

3.6 Case Study: Boiler Control System

A more complicated example involves a real-time software system for the water boiler controller [7]. This problem was originally specified for participants of the competition in the International Symposium on Design and Review of Software Controlled Safety-Related Systems [8]. Later it was used in a similar competition in the International Seminar on Methods for Semantics and Specification [9]. The text of the original description of this problem [7] is 27 pages long and cannot be quoted here for obvious reasons. There is an abbreviated, but significantly changed, 6-page version of the problem description available on the web [10]. Below we produce a one-page summary of the original problem.

**Boiler Water Content Monitor and Control System: Problem Description (after [7])**

The problem is to control the water level in a steam boiler within prescribed limits, to prevent carrying water into the steam flow and cause turbine damage, if water exceeds the upper limit, and prevent boiler from drying out and overheating, if water level drops below the lower limit (Fig. 3-17). There are two measurement devices in the system: water level is sensed by a water content measuring device and steam rate is sensed by a steaming rate measurement device. The water level is controlled by an inflow of feedwater from four feed pumps (with running status indicators), which can be electronically turned on and off. Each pump flow also has a flow monitor device allowing monitor the water flow.

The control law imposed in this problem is the following (known as bang-bang control): if the water level drops below the operating minimum, the feedwater pumps are turned on; if the water level rises above the operating maximum, the feedwater pumps are turned off. If control is lost, the boiler must be shut down. The boiler has an Instrumentation System, which is the only means of communication between the boiler instrumentation (two sensors, feed pumps with status indicators, and flow monitors) and the external controller. All communication between the two is carried via a serial link RS232C.

The message level communication protocol over RS232C is strictly defined. The communication is carried out by a series of transmission units (transmissions). A single transmission is defined as a series of messages bracketed by the ASCII STX character and ASCII ETX character. A transmission may contain zero to fifty messages in any order. Each message is composed of exactly 20 ASCII characters, the first ten being the message identifier, and the second ten being the data. There are 21 different types of messages that can be received from the Instrumentation System and 22 types of messages that can be transmitted to the Instrumentation System (all defined in the original problem description).

The program shall have the following six modes of operation:
1) Computer Self-Test, in which all computer hardware devices and communication link are tested;
2) System Test and Initialization, in which all boiler instrumentation devices are tested;
3) Normal Operation, in which the boiler water content shall be maintained between the lower and upper limit;
4) Degraded Operation, in case the water level is within prescribed limits but one of the instrumentation devices fails, still allowing the boiler to operate correctly;
5) Emergency, in which, despite failure of the water level measurement device, the program shall use all available present and past data to calculate water content on a continuing basis and let the boiler continue operation;
6) Shutdown, in which the program shall emit visual and audible (alarm) signals to the operator announcing reaching this state.

The control program shall be capable of dealing with the following failures:
1) Pump fails to start or stop, or the pump indicator locks on or off;
2) Flow monitor locks into water flowing or not flowing indication;
3) Water level sensor indication is incorrect;
4) Steam rate sensor is incorrect;
5) Corrupt messages and link failures.

The program shall always display the following information: the mode the program is operating in, system time to the five second interval, and status of all components and their individual outputs as “serviceable”, “unserviceable”, or “unverified” along with the associated readings in engineering units. The operation of the program shall be automatic after startup, and the only operator commands shall be “shutdown” and its confirmation.

As described above, this problem is real-time and safety critical, because it includes explicit and implicit timing requirements (on the transmission of data and on the rate of change of water flows, respectively), as well as specific requirements on dealing with failures to assure safety of boiler operation. In this example, we will address only real-time, but not safety, requirements. Developing a software requirements specification document from the original problem description [4] is a daunting task and will not be pursued here. Instead, we will rely on the intuitive understanding of the problem, based on the above description.

The initial step is to derive the context diagram. As indicated in the problem description, as well as on the physical diagram (Fig. 3-17), the control computer does not have direct access to the sensors or actuators. All information about the boiler process can be obtained only from the Instrumentation System via RS232C serial link. The same is true for sending control information to the boiler. Therefore, the context diagram includes only two principal interfaces, one to the instrumentation system, and another one to the operator, as indicated in Fig. 3-18, plus an interface to the database, due to a requirement to track the measurements history. On the context diagram, we additionally indicated the sensors and actuators connected from the boiler to the instrumentation system, to have a better idea what types of signal values the program has to deal with.

The first level decomposition diagram should include data transformations corresponding to each crucial part of the controller functionality. Respective components, shown in Fig. 3-18, include:
- Communication Link, which handles the communication over RS232C and parses messages;
- Shared Memory, which is a data store retaining the most current values from/to the instrumentation system;
- Processing Unit, which is a control transformation determining current mode of operation and associated actions within each mode;
- User Interface, which presents to the screen current status of the boiler and its instrumentation and accepts operator commands;
- Storage Handler, which takes care of storing and retrieving history data.

First level decomposition diagram can be split further into individual data flow and control flow diagrams for the transformation units from Fig. 3-19. Individual functions of units shown in Fig. 3-19 are simple, such as that of User Interface or Storage Handler, or complicated, such as that of a Communication Link, but all of them are straightforward, except that of the Processing Unit. The Processing Unit has to take the burden of all necessary computations to derive appropriate control commands and recognize any failures. Based on the problem description and the context diagram (Fig. 3-18), the Processing Unit interfaces to the following units:
- Shared Memory, to obtain latest boiler data and provide results of computations to control the boiler
- User Interface, to provide display information and receive shutdown commands
- Storage Handler, to obtain historical data, in case some of the measurement devices fail.
It should also include or involve external unit to calculate predictions of boiler (water level) behavior in case it is needed in the Emergency mode.

3.7 Practical Approach to Real-Time Software Design: p-ARS

Based on the above discussions of two design methods, SADT-RT and DARTS, and two case studies, cruise controller and boiler controller, we should now be able to understand the nature of a real-time design, its principal characteristics and basic components. Since the software design is always based on requirements, we must first ask the fundamental question: What is the most essential information in the requirements specification? The answer is as follows: It is the I/O information, that is, understanding of the system interactions with the surrounding environment. Thus, the key issue in beginning the design of software for a real-time system is to have its requirements specification expressed in terms of input/output functions. Such description is a starting point to develop the software architecture.

To look for I/O information in the requirements specification, one may use several different perspectives. Our point of view is to look at the real-time system as a special case of the feedback control system as its general model. This assumption means that we should attempt grouping the I/O information into several categories, based on the interactions with the surrounding environment, that is, the plant, user/operator, computer network, and database. This idea of grouping I/O information into different categories, which later determine the software architecture, follows the fundamental software engineering principle of separation of concerns [12].

Despite some well established concepts and principles of real-time computing [13,14], thus far, there has been very little guidance on selecting real-time architectures, either in the engineering literature or in practice [15,16]. However, when one takes a closer look at Figures 2-7 through 2-10, with explanations in Section 2.4, there is little doubt that one can start and should start designing the real-time software controller from the context diagram similar to those in Fig. 2-2, 2-3, 2-5, 3-7 and 3-17. Moreover, one can state it more firmly that information in the context diagram determines the basic real-time software architecture.

The role of a context diagram in determining the basic real-time software architecture cannot be overestimated. Even though context diagram is a relatively old notational vehicle, it has been well established in real-time software design as the basis for architectural development [1,2,17,18,19]. It is at the context diagram level, where the interfaces between the software and the external environment are defined and developed. For this very reason, the concept of a context diagram is indispensible as a starting point in designing software architecture.

Speaking more precisely, context diagram is a graphical representation of all interactions with the environment shown in the form of inputs and outputs with all their characteristics. One should remember that detailed description of inputs and outputs is critical for proper formulation of the design. They should be defined in terms of some or all of the following characteristics:
- source, destination and direction
- physical form and logical format
- quantity, unit of measure and accuracy
- data type and range of validity
- illegal values, error messages
- timing and frequency
- operator actions, etc.

The application of a principle of separation of concerns leads us, from all context diagrams presented in figures mentioned above, to the point, when it becomes immediately clear that the software architecture must include parties responsible for the following interactions with all external elements:
- inputs from and outputs to the plant (sensor/actuator component)
- interaction with a user (user interface component)
- possible communication with other controllers/processors (communication link component)
- interaction with storage devices (database component)
- enhanced by the processing (computational) capability (processing or computational component), plus
- the time source, which can be internal or external depending on circumstances (timing component).
An example of such design, which can be considered a basic and generic design pattern for real-time systems software, is presented in Fig. 3-20. It is a template of an architecture, which is suitable for designing software for any existing real-time system. The individual units on this diagram can be further designed in detail using any specific design methodology, whether structured or object-oriented. They can be implemented in a variety of technologies, for example: as sequential modules executing on a single processor, as individual concurrent tasks running on a single processor, or as separate tasks run in parallel on multiple processors or even on a distributed system or network.

This architectural pattern is extremely useful, since it addresses a whole variety of issues important in designing real-time software, such as:
- expressiveness, it covers all cases of real-time systems within the framework outlined in Section 2.4;
- scalability, additional units of each category can be added without any change in the architecture;
- extensibility, new software requirements can be introduced without any need to change the basic architecture;
- flexibility, replacing individual components due to conceptual or technological changes is extremely easy;
- reconfigurability, it is possible to map this basic architecture on a variety of hardware configurations;
- retroactiveness, applying this concept to legacy systems clarifies the existing designs and allows reengineering.

To summarize the essential steps of our method, called practical Approach to Real-time Software, p-ARS, let us list its basic steps:
1) Express requirements in terms of I/O information. The result is a list of requirements as a subset of requirements specification document.
2) Map the I/O information onto four basic categories and draw the Context Diagram. The result is a Context Diagram with detailed information on all signals connecting a real-time system to the environment.
3) Derive the Generic Architecture according to Fig. 3-20, identifying all processes (internal units) corresponding to the Context Diagram. The result is an architectural diagram containing respective processes and their interconnections.
4) Determine details of interconnections among the internal process units of the architectural diagram. The result should be a detailed description of each individual connection, in terms of data types and timing characteristics.
5) Proceed with detailed design, using a design method of choice, either structured, such as SADT-RT or DARTS, or object-oriented, which will be discussed in Chapter 4.

To show the effectiveness of this approach, we will now apply the concept of p-ARS to the design of an Automatic Teller Machine (ATM). ATM is a device that should be well known to the reader and is likely to be used as an example in introductory software engineering courses. Although one may be reluctant to admit that it is a true real-time system, its responses must be delivered in real time, otherwise it may cause problems to customers, banks, or other institutions. Therefore, presenting software development for this type of application may be very instructive.

To illustrate the use of p-ARS design method, we will not go through the formal requirements specification, but will assume that the I/O requirements are known and will go straight to Step 2 of the design. To develop the context diagram of the ATM controller properly, we need to identify all inputs and outputs of the ATM machine. This has been done in Section 2.3 and the respective diagram is presented in Fig. 2.2. From this context diagram it is straightforward to identify processes, which will respond to user inputs. This will include CardSlot and KeyPad processes taking care of the customer, but also BankLink process responsible for transferring account information to and from the bank, as well as the Maintenance process responsible for interaction with the service person. Important output processes include Display, which is the heart of the ATM, informing the customer about the progress in transaction (and can be combined with KeyPad into GUI), CashDispenser, taking care of releasing money to the drawer, and Printer and Beeper processes, which are responsible for presenting one-time information at the end of the transaction. The latter three processes are so simple that can be combined into one. This concludes Step 3. All the interactions are partially sketched in Fig. 3-21, with the maintenance process isolated, since it is not a part of regular operation. Completion of Step 4, that is, determination of details of all the interactions is left to the reader as an exercise.

3.8 References