Real Time Software Development – an Engineering Approach
Andrew J. Kornecki and Janusz Zalewski

Chapter 2 - **Specification and Design of Real-Time Software**

In this chapter, we introduce principles and notations for real-time software development. Concepts of software development methodologies, including methods, techniques and tools for real-time systems are presented. Basic aspects of requirements specification for real-time software are outlined. Notions of a physical diagram and a context diagram are introduced, including the major components of the environment. Two major approaches for software design: structured and object-oriented development are overviewed.

2.1. Background

The first thing a student or an engineer has to realize about software development is that programming is not the only, and maybe even not the most important, part of developing software. Writing a program code, that is, coding, is just one of several steps to make software work. What are the other steps?

Before anyone is able to write the code, what is usually needed is a good idea about the program flow, that is, understanding what actions should the program take and in what sequence. In other words, we need to know the structure of the code, how it would look like, before starting to write it. It is not sufficient to have it in the head of the programmer; we need to present it in some non-volatile form. For simple programs, this could be as straightforward as a flowchart, but for real-time programs, which are concurrent, this is not enough, because flowcharts are sequential. There have to be other ways of presenting the logic and flow of control for the most complicated concurrent software. Overall, we need to express, without coding yet, the way "How are we going to solve the problem?" for which the software is to be built. These activities are called the design phase, as opposed to the coding phase, which is developing the program (coding it). The advantage of having the design phase and expressing the program design in a clear and consistent manner is that coding can be extremely simplified and automated, almost left to the secretary, who just needs to fill in the blanks of the design sheets.

However, how to solve the problem can only be determined once we have a clear understanding what it is that we are solving. So the design phase has to be preceded by the problem statement phase, which is technically called requirements specification phase. The requirements specification document, which is the result of this phase, should answer the question "What is the problem we are solving?" So the emphasis of this phase is on the WHAT, as opposed to the emphasis on HOW, in the design phase. A requirements specification document has to present a clear definition of the problem, for which software is to be developed.

The combination of these and other phases of software development is jointly called a **software lifecycle** (or software development cycle), and includes the whole process of software development, from the inception of a concept through the delivery phase and other necessary activities, such as personnel training and even software retirement. It is called a "cycle", because software goes through most of these phases multiple times per lifetime, due to modifications, upgrades, fault corrections, etc., which is jointly called the a maintenance phase. In this book, we only cover a few most important phases of the software lifecycle, shown in Fig. 2.1. The coverage of the entire software development process has been well defined in the IEEE Std 1074 "Developing Software Life Cycle Processes" [1].

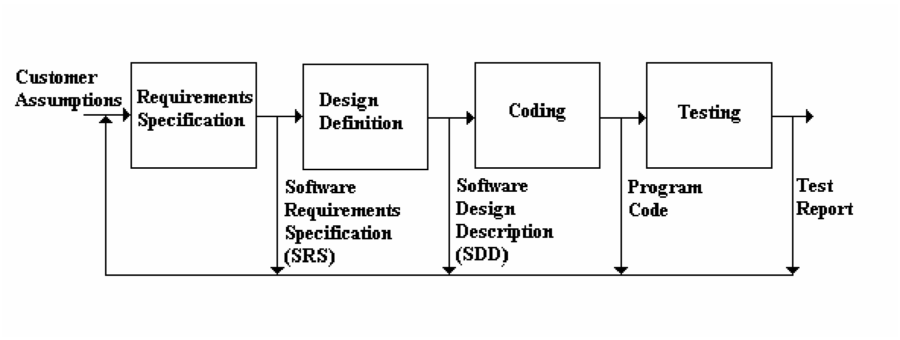
Customer Assumptions
Requirements Specification
Design Definition
Coding
Testing
Software Requirements Specification (SRS)
Software Design Description (SDD)
Program Code
Test Report

Fig. 2.1. Typical phases of the Software Development Lifecycle.

In summary, we will concentrate on four basic phases only. First, we will deal with the **requirements specification**, focusing on how to present well "what is to be developed". In traditional terms, this is called problem formulation or problem description, but for software it is much more than the precise formulation of the problem. The result of this phase is a document called Software Requirements Specification (SRS) developed mostly based on an input from the customer. Next, once we have the understanding of the problem, we will focus on the **design** phase, which usually answers the question "how to develop" what is specified in the requirements. The formal result of this phase is a document called Software Design Description (SDD).

Converting the design into the program is called the **coding** phase, which is the phase directly following the design phase and resulting in a straightforward realization of the design in a form of the code. In addition to the design phase, we will include the **testing** phase, without which none of the software development projects would be complete. The importance of testing should not be underestimated, as it is the phase, which allows performing rigorous and systematic checking of the quality of the software product. The result of a testing phase is usually a Test Report documenting all finding during program testing. It should be noted that modern software development methodologies are proposing that testing and quality control be distributed along the entire development process.

2.2 Principles of Requirements Specification for Real-Time Systems

In the development of requirements for real-time software, the key issue is to understand precisely how a requirement should be expressed, that is, what format it should have. The format is as much important as the contents of a requirement. Without a proper format, requirement cannot be read and comprehended properly by the developer. There are several standards for writing software requirements, one of them probably the most widely used, IEEE Std 754 [2]. They are too extensive to be followed here, so we will give a few rules of thumb, which are constructive enough to give an idea, how a good requirement should look alike.

In the first place, one should realize that the requirements are written to address the developed software, rather than any other entity, such as a user, network, organization, etc. Therefore, the subject of each requirement statement should be the software to be developed. Consequently, a statement such as “A user shall have access to the system 24 hrs a day”, is not a properly formulated requirement and should be rephrased to read, for example, that “The software shall operate 24 hrs a day”.

Secondly, we are writing requirements which software must meet, so they have to be expressed in some compelling fashion. Therefore, a properly constructed requirement typically starts like this: "This software shall [do this, this and that]". In other words, a properly written requirement should include a noun equivalent to "software under construction" and a verb such as "shall", "must", or "has to", making this requirement mandatory.

Furthermore, an individual requirement should be expressed in a single sentence. If it takes more than one sentence to express a requirement, most likely this requirement has to be split into several ones. If there is a need to add something beyond a single sentence, for example some clarifications, this can be made a part of a requirement but should be included under an additional clause, such as a note.

Such note is not a requirement per se, but can be treated as an attribute of a requirement. There is a number of other attributes one can associate with each individual requirement. For example, for obvious reasons, requirements should be identifiable, for example by numbers, and their origins, that is, source, should be stated. A well written requirement can make references to other requirements, but only to those which are preceding it; a requirement should not make any forward references.

Regarding the contents, it is good to group requirements into categories depending on their mutual relationships. Real-time systems are almost always related in some way to the surrounding environment, so it is natural to categorize requirements according to the following scheme:

- input requirements (those related to input signals or data)
- output requirements (those related to output signals or data)
- joint I/O requirements (those related to input and output data simultaneously)
- processing requirements (those related to internal functions of software).

In addition to functional requirements, that is, those related to the functionality of software, such as those enumerated above, there is usually a group of requirements, which can be called non-functional, because they are related to certain attributes of software, rather than to its primary functions. In this group we usually place requirements related to performance, safety, reliability, security, etc. A sample requirement, which illustrates the way requirements are written, is shown in Table 2.1. An example of functional requirements specification for an Air Traffic Control System is discussed in Section 2.5.

Table 1. Sample software requirement.

Requirement 4.3.5.2
The conflict alert function shall provide an alert to the controller, if two or more aircraft are too close in proximity to each other.
Note. The normal separation distances over US air space are: - 5 miles horizontally, and - 1,000 to 2,000 feet vertically.
Source: PS Para6

2.3 System and Its Environment: Physical and Context Diagram

As mentioned in the previous section, the key to developing requirements for real-time software is to understand its relations with the surrounding environment, with which the software is going to interact. The principal ways of expressing these interactions are a physical diagram and a context diagram. A physical diagram represents all the physical entities, including a computer system or systems, which are playing any role in this particular application. These entities are usually sources of data for the software to be developed or destinations of data. Therefore physical sources and destinations of data have to be identified and their interconnections with the computer system determined. Once this is done, the physical diagram can be created.

A context diagram is just a refinement of a physical diagram according to one particular criterion: What is the physical form and logical format of the data being exchanged by our software with the environment? In other words, we need to determine what is the form of data our software is to receive from data sources and has to send to destinations.

The simplest way the context diagram is usually presented is by a circle representing the software and rectangles representing all external devices identified in the physical diagram. Then arcs have to be drawn between the circle and the rectangles, representing respective connections, including directions of data flow and exact descriptions of data formats, frequencies, etc. Exact representation of all signals and data items exchanged between the software and external devices is extremely important, because in terms of these signals and data items usually software requirements are expressed. An example of a Context Diagram for a simple system, such as a microwave oven controller is presented in Fig. 2.2.

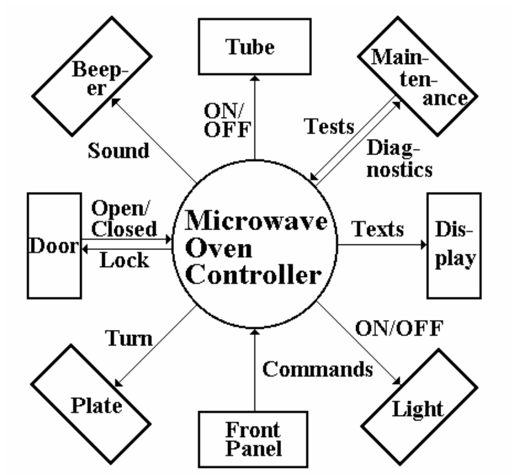

Fig. 2.2. Context diagram of a microwave oven controller.

When the system is relatively simple, producing a physical diagram is not necessary. This is the case of developing software for an automatic teller machine (ATM). In such case, it is very straightforward to produce the context diagram, which depicts all the interactions with the environment. Such an example is presented in Fig. 2.3. To develop this diagram, one has to realize the existence of all the external devices interacting with the ATM software. Search for the input devices, those exclusively providing signals (commands, status, or data) to the controller, without receiving anything back from it, reveals only one such device, a keypad. Output devices, those to which control signals or data are sent by the controller, include: display, beeper, printer, and cash dispenser. Input/output devices, performing both functions, include the following:

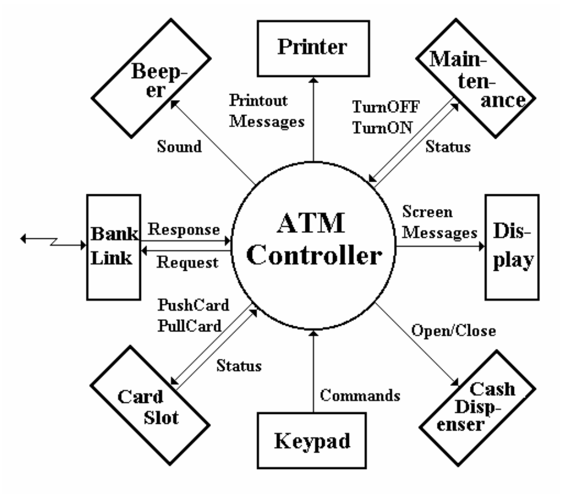

Fig. 2.3. Example of a context diagram for an ATM.

- card slot, to read the credit card and return it after transaction
- link to the bank, providing communication of an ATM unit with the central banking system
- maintenance box, to allow a clerk or a technician to service the unit.

In addition to that, the following issues should be discussed for completeness. There is no deposit box on our diagram, but only because there is no more space for another external device in the Figure. Also, a security camera is omitted, but this is on purpose, because the security system is normally implemented on a different computer system not being a part of an ATM software.

One other important issue needs to be emphasized. In a Context Diagram, all connections between software and external entities have to be clearly defined. This is reflected in Fig. 2.3 by naming all signals from and to external devices. Typically, however, signal names are insufficient and other signal attributes have to be added to provide their full description, such as: kind (status, command, or data), exact data type (text, integer, real, enumeration, etc.), range of valid values, frequency (if applicable), and so on.

In a much more complicated case of developing software for an air traffic control system, it makes more sense to start from a physical diagram. Such a diagram would identify, at least, the following external entities, as shown in Fig. 2.4:

- air space with aircraft subject to manipulation according to instructions from the ground
- air-traffic controllers advising pilots on taking flight decisions
- displays as the major source of information for the controllers
- en-route centers
- flight plans
- weather services, etc.

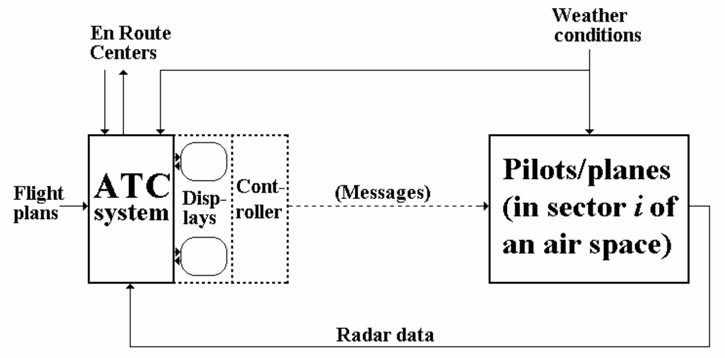

Fig. 2.4. Physical diagram of an air traffic control system.

A corresponding context diagram, based on this physical diagram, would have to identify exact data sources and destinations, and include the following entities external to software, as reflected in Fig. 2.5:

- radars
- controller displays
- flight plan database
- weather server
- communication link to the en-route centers, etc.

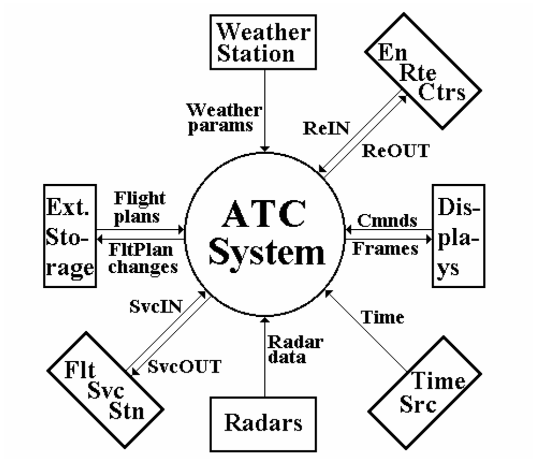

Fig. 2.5. Context diagram of an air traffic control system.

2.4 Basic Categories of Real-Time Systems

Knowing what are the typical layouts of real-time systems may help in identifying the right sources and destinations of data, and create the right physical and context diagrams. We have to realize that the origins of real-time computing lie in feedback control systems, where an analog (and not necessarily electronic) controller was responsible for taking the measured value of a certain quantity from the controlled process, or object (typically called a plant), to compare it with the desired value to compensate for disturbances and produce a control signal. This signal, when applied to the process changes the controlled variable's value in the desired direction to impact the controlled process' behavior. This is reflected in a diagram in Fig. 2.6.

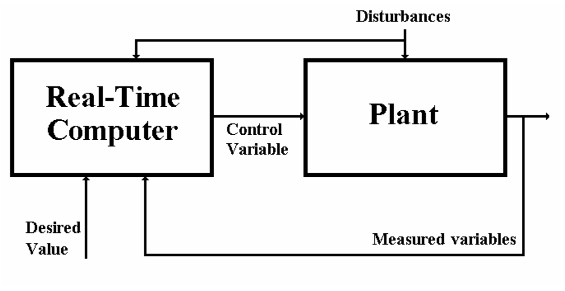

Fig. 2.6. Typical feedback control system.

Such function of the controller means that it has to have on-line connections with the controlled process in both directions, to take input signals from the process and send output signals to the process. With the introduction of digital computers, functions of feedback controllers did not really change but have been significantly extended. Now the controller, usually a microprocessor, in addition to handling input and output connections with the process and performing respective computations, may have a significant number of other interactions with the external world, which roughly fall into three categories, as shown in Fig. 2.7:

- user interface, to let the operator monitor the process' behavior
- communication with other controllers on the network, for example, in a hierarchical system
- database, to store and retrieve relevant information on a permanent medium.

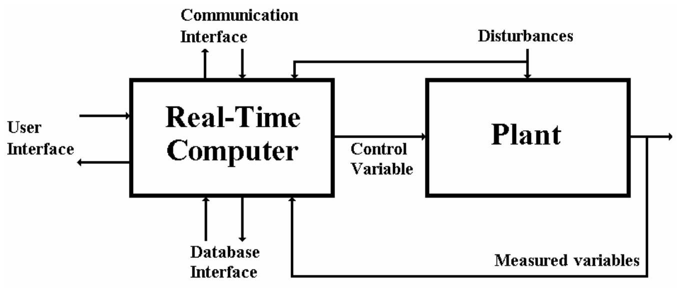

Fig. 2.7. Real-time computer interacting with the environment.

Having established computer's interaction with the process and a broader environment as the central concept in real-time computing, we can now take a closer look at the possible configurations of real-time systems. A typical configuration is that of a data acquisition system, where only measurements are taken into the computer and no control information is sent to the process. This is as shown in Fig. 2.8. Examples of such systems include weather data collection and monitoring, satellite telemetry systems, and, surprisingly, an air-traffic control system, because, contrary to its name, there is no automatic control of aircraft in space.

If we reverse the situation and break the feedback loop, so that no measurements are fed into the computer, leaving only the signals from the computer to the process, we have a well-known example of programmed (open-loop) controller, a situation reflected in Fig. 2.9. Examples of such systems include: microwave oven controller, washing machine controller, etc.

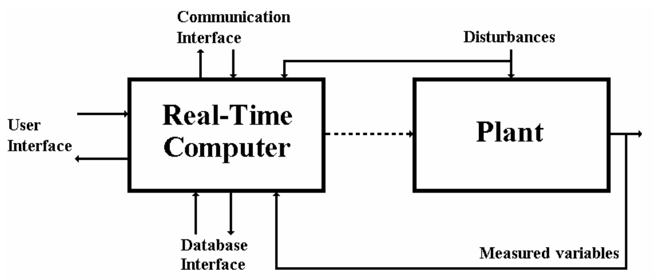

Fig. 2.8. Real-time computer as a data acquisition system.

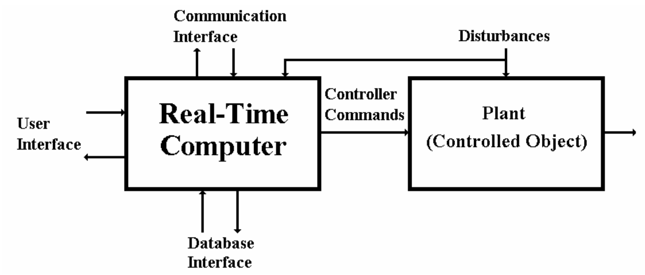

Fig. 2.9. Real-time computer as as programmed (open-loop) controller.

Now, what happens if we break both connections of a computer with the external process, is it still a real-time system? Yes, it certainly is, if it meets the definition of a real-time system: a computing system with bounded response time. Note that in previous two cases, data acquisition systems and programmed controllers, the requirement of "bounded response time" is mostly the result of a necessity to provide timely response to external events, coming from the external process. In a new type of system, when the computer is basically disconnected from the process, the timing requirements come mainly from other types of external interconnections (Fig. 2.10). For example, such a reduced real-time computer may serve the purpose of a real-time distributed simulation, if the communication requires timely responses from the network. Similarly, if the disk/database traffic requires a timely response, then usually a real-time multimedia system is in place.

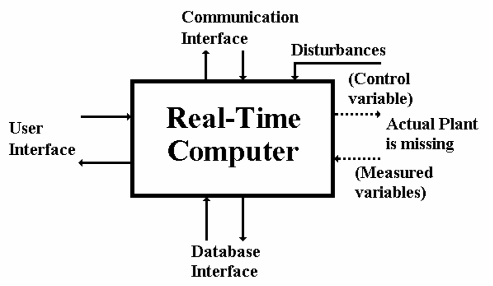

Fig. 2.10. Real-time computer functioning autonomously, without plant.

To summarize, from the application perspective, there are four different types of configurations of real-time systems:

- full feedback control systems, such as a cruise control system in a car, nuclear reactor control or flight control system
- data acquisition systems, such as data loggers or telemetry systems
- programmed controllers, such as a microwave oven controller or a washing machine controller
- reduced real-time systems, such as real-time multimedia systems or real-time simulation systems.

Knowing the type of the system to be developed helps in defining the requirements on software as well as in choosing the right software architecture, which will be discussed in the next chapter.

2.5 Software Requirements Development: Air Traffic Control System Case Study

To illustrate the way requirements are presented, we will discuss a case study of an Air Traffic Control System (ATCS). An academic version of this problem originated at the 6th Workshop on Parallel and Distributed Real-Time Systems, held in Orlando, Florida, in 1998. Participants were given a Problem Statement, written in plain English, related to the development of an ATCS for the US airspace. It has been formulated with help of the Naval Surface Warfare Center of the US Navy and represents a very practical project. The task, as such, is extremely complex and nearly unsurmountable to student teams, for which it was meant. However, we decided to include it in this book, because it is very realistic and reveals several important issues and challenges in the specification, design, implementation and testing of real-time software.

The original Problem Statement, as it was distributed to the participating teams, is presented below. The text has not been changed, however, its organization has been adjusted to meet the needs of this book. In particular, the sections and their headings have been added.

Problem Statement for the Sixth Workshop on Parallel and Distributed Real-Time Systems

Preamble

Air traffic control presents many challenging problems in distributed real-time computing. The August 1997 issue of IEEE Spectrum features an article that describes some of these challenges [T.S. Perry, In Search of the Future of Air Traffic Control, Vol. 34, No. 8, August 1997]. Thus, air traffic control has been chosen as the problem to be solved for presentation during a special session of The Sixth Workshop on Parallel and Distributed Real-Time Systems (WPDRTS). Solutions to the problem statement should be in the form of a design, a prototype or an architecture, and should provide sufficient detail to be critically evaluated. A session at WPDRTS will be dedicated to the presentation and discussion of the solutions.

This problem statement has been developed in coordination with engineers who are involved with software upgrades to the US air traffic control system (ATCS). Additionally, a US air traffic controller was involved in producing the problem statement. Thus, the problem statement reflects characteristics of the US ATCS.

Background
The United States is divided into 20 airspace regions called Air Route Traffic Control Centers. Each Center is further divided into sectors. An air traffic controller can control one or more air traffic sectors in a Center. The air space surrounding an airport is termed the TRACON (Terminal Radar Approach Control) area. The TRACON area is usually defined as a 40 mile radius around a major Airport, having an altitude of around 10,000 feet. The TRACON receives control of aircraft that are landing at the TRACON's airport and passes control of aircraft that are leaving the TRACON airspace. The Tower has final approach control of an arriving aircraft and departure control of aircraft wanting to leave the airport.

The specific problem to be solved for WPDRTS is to design an air traffic control system that is fault tolerant and scalable. The requirements of ATCSs include real-time aspects. Due to the criticality of ATCSs, the system must continue to provide full functionality, even though some hardware processors or networks may fail. The system must be able to handle varying loads. As a minimum, it must be capable of handling 2600 flight plans and 700 active tracks in one Center airspace region (these terms are defined in [1] and also in the following paragraphs). The ATCS must also be able to handle loads that exceed this minimum. Furthermore, there is no upper bound (or worst case) load, and there is significant variance in loading, so the average load is virtually useless characterization.

General Overview
Functionally, the ATCS must be able to provide track prediction, conflict probe and alert, resolution advisories, and minimum safe altitude advisory warnings, as well as time-based arrival and departure metering. The system must be able to display the available information (track, flight plan, metering lists, arrival lists, departure lists, flight strips, and weather data) to up to 120 displays and allow various controller inputs to manipulate not only the display but also the track and flight plan information. This system must be able to send and receive data to and from surrounding Centers, TRACONs, and Towers. Finally, this system has to be able to provide data for recording of any sent or received data between processes and/or remote systems, as well as data while any process is running. These requirements are detailed in the remainder of this document.

Data Sources
This system must be able to acquire data and convert it into a format the controller can use. Data inputs include a maximum of 25 long range and short range radars, with new radar or track information arriving every 5 seconds. Multiple radars can return radar hits on the same aircraft. From these multiple radar hits a primary and secondary radar must be chosen for each aircraft. The radar data from all different radars in the system must be mosaiced so that all the radar is projected onto a single plane of reference. The track information consists of an aircraft's (x,y) coordinates, altitude and beacon code. The beacon code is a unique numeric identifier that only one aircraft can have in a particular Center airspace at any instant. This system must be able to handle 2600 flight plans that can be input from other Centers, flight service stations, and bulk flight plan storage (tape storage). Flight plan data consists of aircraft identification, aircraft type and equipment, beacon code, filed airspeed, coordination fix, coordination time, requested altitude, and route of flight. Once the new track information is received, it must be correlated with the flight plans stored in the system. Finally weather information must be accepted so that it can be used later for track prediction and to be displayed to the controller.

Primary Functions
Several alert and advisory functions must be provided so that safe separation of aircraft in the Center airspace is provided. This system must provide a track prediction function as well as conflict probe, alert, and advisory function. The conflict alert function will provide an alert to the controller if two or more aircraft are too close in proximity to each other. The normal separation distance over US airspace is 5 miles horizontally and 1,000 to 2,000 feet vertically. The track prediction function will provide a future track position of a given aircraft. The future track predictions of all aircraft must be examined to determine if any of the aircraft currently under control in the Center airspace will be in conflict with each other in the immediate future. The resolution advisory will provide the controller with a possible solution to the current and future conflict. Finally, a minimum safe altitude warning function must be provided to determine if an aircraft is flying in close proximity to any type of natural or man-made obstructions in the airspace. If an aircraft is flying at an unsafe altitude, the controller should be alerted, so that an action can be taken to avoid a possible collision.

A time based metering function must be provided by this system to optimize the arrival and departure aircraft flows and create a plan that satisfies the airport arrival rate restriction as well as increase fuel efficiency and reduce delays. The time based metering functionality must make use of performance models of all aircraft that fly in the Center and TRACON regions and adapt to changes in the air traffic situation, controller imposed constraints, and pilot and airline preferences. Metering lists will be provided to the appropriate controllers and the controllers will have the ability to manipulate the lists according to their aircraft sequence requirements.

Displaying Information
This system must be able to handle up to 120 separate controller displays, that will display Center and sector boundaries, arrival and departure routes, aircraft positions, aircraft data (including aircraft id, reported altitude, assigned altitude, ground speed), metering arrival and departure lists, weather information, as well as electronic flight strips. A flight strip consists of data the controller needs to know about the aircraft in order to control it. The flight strip contains at a minimum the following data: flight identification, aircraft data, true airspeed, estimated ground speed, sector number, computer identification number, strip number, previous posted fix, assigned altitude, coordination fix, coordination time, beacon code, and any remarks. The controller must be able to update and modify the flight strip and these changes must be reflected on any other flight strips that may be displayed in the system. The displays must be able to accept controller inputs that modify the flight plan information and track data, as well as provide the ability to hand-off control of an aircraft from one sector to another sector.

Communication and Recording
Center-to-Center and Center-to-TRACON communications must be provided so that flight plan data and track data can be passed to other Centers or TRACONs that may be receiving arrival flights or handing off departure flights. This communication link can also provide the TRACONs with the ability to talk to one another when there are flights that are going from TRACON to TRACON and are not entering Center airspace. This link can be also used to provide arrival and departure metering lists to adjacent centers as well as any flight plan or track information any Center may want to request.

This system must provide a comprehensive data recording facility that allows performance analysis of the system and data recording reductions to provide specific information (flight, track, display, and controller input) about any and all aircraft, plus information about internal process functionality, interprocess communication and inter-Center and inter-TRACON communications.

Conclusion
The required ATCS is a “dynamic” real-time system. Its loading will vary significantly over time, and has no upper bound. Loading scenarios can vary significantly, hence average loading of the ATCS is not a highly useful metric for schedulability and other analyses. Although an upper bound could be possible imposed artificially, this may not be a cost-effective solution, since preallocation of computing resources for such a worst case would lead to very poor resource utilization. A dynamic resource management policy is thus preferred.

This problem statement has to be converted into a rigorous and systematic Software Requirements Specification (SRS) document. This may be difficult, if not impossible, unless the developers have clear understanding of the problem. What may help is grasping the terminology specific to the application domain. For the purpose of this book, a glossary of basic terms from air-traffic control is provided in Appendix A.

There are several possible formats, in which the SRS document can be expressed, one of them, probably most frequently used is the IEEE Std 830 [1]. For the purpose of our discussion, this standard is too comprehensive, however, and we will use a simplified template as illustrated in Table 2-2. It includes four items, called attributes of a software requirement:

- reference number (and possibly a name) of the requirement
- actual contents of the requirement
- additional information, such as a note, which is technically not a part of a requirement, but may help in understanding or applying it, and
- the sorce from which the requirement originated.

Sometimes it is necessary to trace requirements modification, so a fifth attribute is also added in the template.

Table 2.2. A simple template for writing software requirements.

Reference number (and possibly a name) of the requirement
Actual contents of the requirement. This is the basis for producing software to which the developers must adhere to. Must be expressed in a single sentence including one of the verbs " shall" , " must" , or " has to" applied to the software to be developed. Must be placed on software not on anything else (user, environment)
Additional information, which is technically not a part of the requirement' s contents, but may help to understand, clarify or apply the requirement. This includes use case diagrams & descriptions.
The source from which the requirement originates.
Any modification history of this requirement.

Based on the problem statement presented above, as well as on additional studies of available air-traffic control literature [3-15], an SRS document has been developed, which is presented in Appendix B.

2.6 Verification of Requirements Specification

Once the requirements specification document has been developed, it has to be verified, just like any other product in the software development process. There is a number of verification techniques one can use in this activity, such as formal reviews, inspections, walkthroughs, etc. What is truly important in all these techniques, is the criteria one applies during the verification process.

It has been the authors' experience that the following criteria serve well the purpose of verifying requirements: clarity, consistency, completeness and correctness, plus traceability and testability. Just to memorize them better, we call them C^4T^2 (or **four+two)** set of criteria. Definitions of these criteria understood as software attributes, plus examples of use, are given below. They are applied to the software requirements specification of the ATCS. Using these criteria, a number of defects in the original problem statement have been found. They are described in the following format:

- source
- problem reported
- corrective action.

2.6.1 Completeness

Completeness is the property of the specification document that ensures inclusion of all the information necessary to develop a specified system.

Source. Problem Statement.
Defect. The following terms were left undefined: active track, aircraft sequence requirements, air speed, arrival list, arrival metering, coordination fix, coordination time, departure list, departure metering, dynamic resource management policy, flight service station, metering list, resolution advisory, sector, tower, track prediction, weather information.
Action. Define those terms found in glossaries. Leave remaining terms to designer's interpretation.

Source. Problem Statement.
Defect. In the flight strip and flight plan, there are no units specified in which the information is to be handled, e.g. speed: knots, miles/hr, km/hr, etc.
Action. Left to the designer.

2.6.2 Consistency
Consistency is the property of the specification document which ensures that there is no contradictory information in this document. In fact, consistency should also mean lack of redundancy, because a single concept explained more than once is likely to be misinterpreted.

Source. Problem Statement.
Defect. "1,000 to 2,000 feet vertically" is not clear on its meaning, which is right 1,000 or 2,000? Or they are both right depending on circumstances?
Action. Left to the designer (depends on altitudes).

2.6.3 Correctness
Correctness is the property of the specification document that ensures this document's compliance with external requirements, such as related documents, standards, scientific knowledge, common sense, etc.

Source. Problem Statement.
Defect. The phrase "The track prediction function shall provide a future track position of a given aircraft" is incorrectly stated. This is because the future cannot be determined only estimated.
Action. Modify the requirement to include: an estimate and accuracy.

Source. Problem Statement.
Defect. No upper bound on load implies infinite memory requirements, which is incorrect (real memory size must be finite).
Action. Include upper bound on load.

2.6.4 Clarity
Clarity is the property of the specification document that ensures its understanding and non-ambiguous interpretation by the reader proficient in the specification language.

Source. Problem Statement.
Defect. "Data inputs include a maximum of 25 long range and short range radars." Does it mean 25 of each type of the radars or 25 of both types of radars. Amount of radars is not clear. 25 long range and short range radars can be interpreted as 50 radars or 25 radars total.
Action. Resolved by adding the word "total" to the end of sentence. This means 25 of both types of radar.

Source. Problem Statement.
Defect. The system to be developed is termed "system", in most cases, or "ATCS", in some cases.
Action. The term used throughout this Software Requirements Specification has been unified to "ATCS".

2.6.5 Traceability
Traceability is the degree to which a relationship can be established between two or more products of the [software] development process, especially products having a predecessor/successor or master/subordinate relationship to one another.

For this SRS, there is only one level of traceability, that is, all requirements are derived from the problem statement. Therefore, no significant defect have been expected or found. The developers must remember, however, to produce the Traceability matrix, that is, a cross-reference list, which relates each requirement in the SRS (by number) to its origin in the Problem Statement (by some identifier, such as a paragraph number).

2.6.6 Testability

Testability is the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.

Source. Problem Statement.
Defect. Time-based metering need a formula which is a function of aircraft flows, arrival rate, fuel efficiency, and weather before we can test this.
Action. Left to the designer.

Source. Problem Statement.
Defect. "... format that the controller can use" needs to be defined. It is impossible to test that the ATCS has converted the data into a format the controller can use since this format is not further specified.
Action. Left to the designer.

2.7 Software Development Methodologies

In all models of the software development cycle, the software specification case is always followed by the design phase. This is true, no matter whether a model encompasses a full range of activities, as described in IEEE Std. 1074 [2], or a limited subset composed of the four primary phases, specification, design, coding, and testing, as in our case. Once the requirements have been frozen, which means that no more significant changes to the requirements document can be expected, the developers should be ready to start the design process. At this stage, one has to organize the design activities in some systematic manner by choosing an appropriate notation for this phase, techniques to derive a specific representation of software from a more general one, and automatic software tools to assist in the derivation process. A complete set of these three elements (a method, techniques, and tools) is called a software design (or development) methodology.

Simplifying things a little, one can say that the design activities usually concentrate on two aspects:

- architectural design, which for the most part describes the structure of software
- detailed design, which provides the insight into the functioning of the structural elements of software developed at the architectural level.

At the architectural design level, structural units of software (modules, components, etc.) are introduced along with respective interconnections. At the detailed design level, these modules or components are filled with detailed information on specific actions performed internally by each module during its lifetime.

There are two primary approaches to develop the software architecture: structured approach (also called a functional approach) and object-oriented design approach. The difference between the two is significant but, in fact, both approaches have common origin. The common roots and drastic differences become clear, if we look at the operation of software from an abstract perspective. Since their inception in the 1940s, computers and their software, no matter which level we consider, microcode, processor instruction level, operating system, programming language, etc., involve only two primary entities: data and operations on these data.

The way the structured (functional) approach treats these two entities is completely opposite to the way they are treated in object-oriented approaches (Fig. 2.11). In structured approaches, the primary focus is on operations. This means that in the structured development process, we first determine the operations to be executed and their order, encapsulate those operations in a form of a module, function, procedure, subroutine, etc., and then add the data paths in the form of parameters or arguments, through which data can flow in and out of these design units. This is an operations-centered approach.

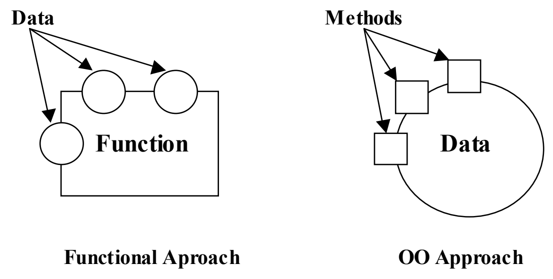

Fig. 2.11. Principal difference between the functional and object-oriented approach.

In contrast to that, an object-oriented approach is data-centered. This means that in the development process, we first determine what data objects are going to be used and consequently design functions, technically called methods, to access these data objects. To illustrate both approaches graphically, let us assume that squares represent operations and circles represent data, as in Fig. 2-11. In a structured approach, a big square would represent respective operations encapsulated in a module, and little circles on the boundaries of this square would represent data transferred as arguments to the calls accessing operations in the square. In an object-oriented approach, a big circle would represent all data items encapsulated as a structure, with little squares spread over the boundaries of this square that would represent methods used to access these data items.

Because of this principal difference in both approaches, respective notations for software design are different. The primary notation used in structured software development is based on data flow diagrams. The notation adopted for object-oriented development is based on class and object diagrams.

Based on this primary distinction, there have been many approaches to real-time software development, which claimed to be methodologies, but in fact they were just methods or specific notations. Historically, one can mention several methods and their notations, which have been used in real-time software design, for example:

- SCOOP (Software Construction by Object-Oriented Pictures) [18]
- MASCOT (Modular Approach to Software Construction, Operation and Test) [19]
- HOOD (Hierarchical Object-Oriented Design) [16]
- DARTS (Design Approach to Real-Time Systems) [17].

None of these methods, however, gained widespread popularity, because they were lacking techniques of well-defined transformations and, in particular, automatic software tools to help in the development process. In addition to that, even though these notations capture quite adequately various intricacies of real-time software, they are not suitable for expressing timing requirements. We will discuss selected development methodologies in the following chapters.

2.9 Characteristics of Automatic Tools for Real-Time Software Development

Contemporary real-time software development is too complex to be done manually by a single individual. Developers are expected to have technical knowledge going far beyond programming languages and focusing more and more on specific software design methodologies and support tools the industry uses on a daily basis. Typically, the use of software tools in support of the development process follows some kind of a software life cycle model, for example, a waterfall model. This means that a design tools is normally expected to support the process in a *vertical dimension* by addressing the next and previous phase of software development, that is:

- support the code generation and prototyping, in particular, for specific programming languages and operating system kernels, and
- support the design verification against the requirements (for example, to assure one-to-one correspondence of design components with the requirements).

There are, however, other aspects related to the way software development tools are used. One of the most important ones is the *environment aspect*. An IEEE standard on tool interconnections addresses this issue [20], and distinguishes among several contexts, in which the tools are used (or interfaces for the tool use):

- user context, which is referring mostly to the interaction via the human-computer interface
- organization context, that is related to the processes, in which the tool is used
- other tools, that is, other software systems with which the tool must exchange data
- platform context, understood as an infrastructure on which the tool is hosted.

Based on this general distinction, one can specify more accurately the various roles a software development tool is playing in the environment, and list corresponding views of the tool, or tool's dimensions, which may be helpful to the developer. In this respect, a user context represents an internal view, or *internal dimension* of the tool, since the task of the user is normally to develop the graphical representation of the design and check it, for example, using tool's animation or simulation capability. These are internal tool functions.

Two other contexts, organization interface and interface to other tools, both refer to the communication the tool has with other processes (projects) within the organization, as well as to communication with other software within the same project. The first type of communication is mostly static (off-line) and relies on exchanging design models with other projects. The second type of communication is more dynamic (on-line or even real-time) and relies on maintaining direct connectivity with other tools. Since it normally relies on peer-to-peer interaction, a corresponding view of the tool involving communication with other tools can be called its *horizontal dimension*. Similarly, since the exchange of design models between projects happens across the organization, it can be called a *diagonal dimension*, symbolizing crossing the project boundaries (or even organizational boundaries). The final context, the platform context, although it is an important factor in the tool environment, can be considered an *external dimension*, and not necessarily critical with respect to real-time software development.

Consequently, for the tools to be fully useful in software construction, four different aspects of the design tool use can be distinguished (Fig. 2-12), referring to four specific dimensions of the tool, with respect to its role in the process and the environment:

- vertical dimension, related to the process aspects of the tool use, specifically to its support for the next and previous phases of the development process
- internal dimension, related to user aspects of the tool (developing the model of the design and animating or simulating its operation and performance)
- horizontal dimension, related to the environment aspects in a view of on-line communication with other tools, for instance, via TCP/IP protocol, and
- diagonal dimension, referring to the ways of exchanging design models with other processes or projects, for example, importing a non-UML model into a UML-based tool.

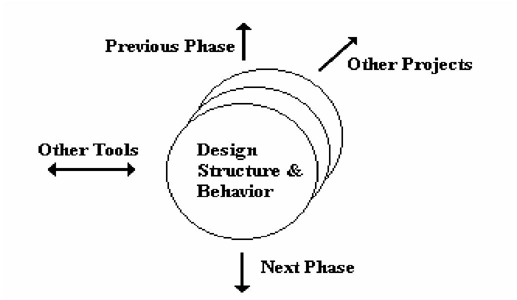

Fig. 2.12. Four aspects of software design tools.

The advantage of this view, from the perspective of tool functionality, is that having the tools operational in all four dimensions, one gets the following properties addressed with respect to both the software product and the software process:

- correctness, completeness, consistency and performance in the internal dimension
- interoperability and connectivity in the horizontal dimension

- testability and traceability in the vertical dimensions
- reusability and portability in the diagonal dimension.

2.9 References

[1] IEEE Std 1074-1997 “Standard for Developing Software Life Cycle Processes”, IEEE, New York, 1997
[2] IEEE Std 830-1993 “Recommended Practice for Software Requirements Specification”, IEEE, New York, 1994
[3] Benel R.A. et al., Advanced Automation System Design, Proceedings of the IEEE, Vol. 77, No. 11, pp. 1653-1660, November 1989
[4] Debelack A.S. et al., Next Generation Air Traffic Control Automation, IBM Systems Journal, Vol. 34, No. 1, pp. 63-77, January 1995
[5] Perry T.S., J.A. Adam, Improving the World's Largest, Most Advanced Systems, IEEE Spectrum, Vol. 28, No. 2, pp. 22-36, February 1991
[6] Perry T.S., In Search of the Future Air Traffic Control, IEEE Spectrum, Vol. 34, No. 8, pp. 18-35, August 1997
[7] Pozesky M.T., M.K. Mann, The US Air Traffic Control System Architecture, Proceedings of the IEEE, Vol. 77, No. 11, pp. 1605-1617, November 1989
[8] Billings C.E., Aviation Automation: The Search for a Human-Centered Approach, Laurence Erlbaum Associates, Mahwah, NJ, 1997
[9] Brenlove M.S., The Air Traffic System: A Commonsense Guide, Iowa State University Press, Ames, IA, 1987
[10] Field A., International Air Traffic Control: Management of the World's Air Space, Pergamon Press, Oxford, 1985
[11] Garrison P., How the Air Traffic Control System Works, TAB Books, Blue Ridge Summit, Penn., 1979
[12] Gesell L.E., Air Traffic Control: An Invitation to a Career, Coast Aire Publications, Chandler, Ariz., 1987
[13] Hopkin V.D., Human Factors in Air Traffic Control, Taylor and Francis, London, 1995
[14] Nolan M.S., Fundamentals of Air Traffic Control. Second Edition, Wodsworth, Belmont, Calif., 1994
[15] Wickens C.D., A.S. Mavor, J.P. McGee (Eds.), Flight to the Future: Human Factors in Air Traffic Control, National Academy Press, Washington, DC, 1997
[16] A. Burns, A. Wellings, HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems, Elsevier, Amsterdam, 1995
[17] H. Gomaa, Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley, Reading, Mass, 1993
[18] G.W. Cherry, Software Construction by Object-Oriented Pictures, Specifying Reactive and Interactive Systems, Dorset House, New York, 1990
[19] H.R. Simpson, The MASCOT Method, Software Engineering Journal, Vol. 1, No. 3, pp. 103-120, 1986
[20] IEEE Std 1175, Trial-use Standard Reference Model for Computing System Tool Interconnections, IEEE, New York, August 1992.