## Module 11

## Real-time Software Engineering; Formal Specification

**Objectives:**

1. Know what a real-time system is.
2. Know what an embedded system is.
3. Be aware of some of the design issues unique to real-time and embedded systems.
4. Know what a formal specification is and when it is typically used.
5. Be familiar with several methods used for generating a formal specification.

**Reading:**

*Sommerville,* Chapter 21.

Web Chapter 27 (Formal Specification, available on the companion website: http://iansommerville.com/software-engineering-book/files/2014/06/Ch_27_Formal_spec.pdf).

**Assignment:**

No assignment for Module 11.

**Quiz:**

No quiz for Module 11.

**Embedded Software**

The term **embedded software** generally refers to software that has been developed specifically to reside in the hardware of some device or machine, and which performs functions specific to that device or machine. Embedded software is not general purpose software, nor is it software that can be installed on just any type of system. Quite often, embedded software is permanently hardwired into the circuitry of the device, and cannot be altered or upgraded. In some cases, limited modification or upgrading is allowed, provided the device uses the appropriate hardware. Embedded software can be found in modern kitchen appliances, phones, automobiles, vending machines, televisions, and a wide array of other electronic devices. Embedded software is usually used in situations where real-time performance is essential, or desired. For this reason, embedded software and real-time software are often considered synonymous, but they are distinct entities. Real-time performance demands not only that the software function correctly, but also that it performs its functions within a specified time window. If a real-time system produced the correct behavior, but did not do so within the specified time window, it would be considered a failure. Therefore, delays caused by using software on a typical computer that is running other, unrelated software simultaneously cannot be tolerated.

Some of the key differences between real-time and/or embedded software and other types of software include:

1. Embedded software typically has to abide by stricter physical constraints that are more relaxed for other types of software. The size of the device in which the software will reside may limit the amount of memory available to the software, since the smaller the device, the fewer chips can be installed. Power consumption will often be limited, especially for battery-operated devices.

2. Direct device-to-hardware interactions may need to be designed into an embedded system, since the use of intermediary device drivers may not be feasible.

3. Since a device containing embedded software may be located virtually anywhere, and be subjected to all kinds of unpredictable use/abuse, the software must be designed to accommodate both normal and abnormal usage.

4. Many embedded systems are expected to run continuously, and depending on the application they may not be allowed to be brought down even for maintenance. Therefore, reliability and dependability play key roles in the design of these systems, and can involve integrating redundancies into the system and data, as well as disaster recovery procedures. Should an update or bug fix be required, it may have to be performed while the system is in operation.

**Embedded Software Design Issues**

Because of the physical constraints and performance requirements imposed on an embedded system, the design process for embedded software must be altered to address these issues. Sommerville emphasizes the following issues:

1. <u>Selection of execution platform</u>

   The hardware and real-time operating system selected must be able to support the required functionality and performance constraints.

2. <u>Identification of stimuli and responses to stimuli</u>

   The behavior of a real-time system can be defined in terms of the stimuli it must be able to receive and to which it must respond. There are two general categories of stimuli: **periodic stimuli**, which occur at regular time intervals, and **aperiodic stimuli** which do not occur at regular time intervals. All the stimuli that must be processed by the system should be identified, and the correct responses determined.

3. <u>Analysis of timing requirements</u>

   This pretty much explains itself. Since real-time systems have specific time windows that must be obeyed, the desired response times must be elicited and analyzed for feasibility.

4. <u>Process architecture design</u>

   “Process” here refers to executing system processes, not the software design process. Real-time systems often have multiple processes running concurrently; the architecture used to construct how these processes are structured, initiated, and managed must be capable of satisfying all system requirements.

5. <u>Algorithm selection and/or design</u>

   Since timing is critical in real-time systems, and since space is limited in embedded systems, the algorithms used must be as efficient as possible both in terms of running time and space usage. Depending on the particular requirements, it may be necessary to design custom algorithms to fit specific hardware configurations.

6. <u>Data design</u>

   The same reasoning that applies to algorithms also applies to data structures used for storing data in memory—they must be optimized for efficiency in terms of running time and memory usage. The structures used to store data must also be capable of being controlled by mutual exclusion control mechanisms, since several concurrent processes may be attempting to access or modify data at the same time, or in any order. The circular buffer example Sommerville describes is a fairly standard example. In this scenario, there may be one or more producer processes that add new data to storage, and one or more consumer processes that retrieve data from storage. There are two key issues that must be handled:

   a. The consumer and producer processes must be managed so that consumers are only allowed to operate when there are data to be retrieved, and producers are only allowed to add data when there is sufficient space in the buffer.

   b. Consumers and producers must not be allowed to access the same location in the buffer simultaneously.

7. <u>Process scheduling</u>

   In addition to designing an optimal process architecture, it is also necessary to ensure that processes are started and terminated at the appropriate times, and that they are executed in the proper order.

Because the design of real-time and embedded systems are so heavily dependent upon specific hardware configurations, it is usually not feasible to use a top-down design approach, where the overall system is decomposed into smaller units. Instead, the low-level details usually need to be addressed first, and the rest of the system built from the bottom up.

## Real-time Operating Systems (RTOS)

The operating system on which a real-time system is executed must be designed to accommodate the timing constraints of the real-time system. usually this means standard operating systems like Windows are not used. A real-time operating system is usually designed to be a barebones OS, containing only those features essential for the proper functioning of real-time systems. Sommerville describes the following components commonly found on most real-time operating systems:

1. A **real-time clock**, used for handling responses to periodic stimuli.

2. An **interrupt handler**, which is used to handle aperiodic stimuli.

3. A **process scheduler**, which is responsible for determining the order of execution of the various processes of the real-time system, and whether or not one process should be allowed to pre-empt a currently executing process.

4. A **dispatcher**, which performs the actual starting of processes scheduled for execution.

5. A **resource manager** for allocating appropriate levels of resources to processes that have been scheduled for execution.

### Implementation of Embedded or Real-time Systems

The key issue here is the choice of programming language. Not all programming languages provide the necessary capabilities for accessing hardware at a low level. Limited space requirements require that the final executable code be relatively compact. Solutions that are easy to visualize and write using object-oriented languages may be too bloated and slow for an embedded system. Most embedded software that must reside on hardware with limited memory and processing power continue to be written in C. There is a version of the Java language that has been developed specifically for use in implementing real-time software, but it tends to be used primarily on systems that have more resources available. One of the key criticisms of using Java for real-time systems has been the lack of control over how the garbage collection feature works. Although improvements continue to be made, it remains to be seen whether Java can unseat C as the language of choice for real-time system implementation.

### Testing of Embedded and Real-time Systems

Verification of software that will be embedded in a device presents several unique challenges. Since the device in which the software will be embedded may be costly, it is better if the software could be tested before being embedded. Unit testing can still be performed as it is with software that is not embedded, but black box testing can only be approximated. Simulator software can be written to try to provide as realistic an environment as possible, but the results will still only be simulated results. At some point the software must be tested in its actual environment.

### A Few More Words Regarding Dependability

Even software that has been thoroughly tested may experience problems during execution. Environmental conditions may cause "hiccups" where improper behavior results not because of an underlying fault, but simply because there is always some degree of uncertainty involved when dealing with any kind of physical artifact. You may recall from a previous course in computer organization that even though a bit is considered to have only two possible states (0 or 1, off or on), what actually defines the "0" and "1" states is voltage; at or above a certain threshold voltage is considered to be "1" while anything below that threshold is considered to be "0". So the actual voltage present can vary at different times, but the use of a threshold value divides the possible range of voltages into two partitions. A sudden power surge or voltage drop can therefore render a correctly working system inoperative.

Since embedded software cannot be modified as easily as other types of software, it may be necessary to build fail-safes and redundancies into an embedded system in order to ensure continued operation, particularly if the software is part of a critical system. An excellent example is the embedded software onboard the rover exploring vehicles that have been sent to Mars in recent years. Since there is no way to know for sure what conditions the rover may encounter, redundant backup systems are essential. If one system should fail for some reason, the backup system may be able to take over. Remote systems also require built-in features to allow for diagnostics, or to set or reset the values of certain attributes. A watchdog timer can be implemented as part of the system to allow the system to perform its own diagnostics or system reset, should the system become inoperable. Under normal operation, the watchdog timer is periodically reset, or "stroked". Should the timer expire, as in the case of a malfunction, the timer can trigger the execution of a diagnostic and repair sequence, or a complete system reboot.

## Formal Specification

Formal software specification has always been an intriguing concept. Virtually every other engineering discipline relies heavily on the use of mathematics to formally guide the engineering process. Civil engineering, for example, depends on excruciatingly detailed analyses of materials characteristics, physical forces, etc., to ensure that a building will not topple during a storm, or that a bridge will not collapse if a fleet of fully-loaded trucks should happen to be crossing in tight formation. Chemical engineering relies on the ability to formally and accurately determine everything that occurs during a chemical reaction: what compounds are consumed, what compounds are created, how much energy is released or consumed in the reaction, etc. Electrical engineers must employ numerous formulae to formally specify how to build the various electronic components that are so pervasive in today's society.

Oddly, this reliance on mathematical rigor has not become an integral part of software engineering, even though intuitively it is reasonable to assume the use of such rigorous methods should lead to higher quality software with fewer faults. There was a movement, of sorts, towards increased use of formal methods several decades ago, but the strength of that movement has all but died out in recent years. Sommerville's decision to relegate his chapter on formal specification to an online-only chapter is indicative of the decreased importance now given to formal specification. In his chapter, Sommerville offers four reasons why formal specification never caught on:

1. Quality has been achieved in software development via other means, such as structured programming, the principle of information hiding, software reuse, and design patterns, to name a few.

2. A shift in priorities in the software industry that has made time-to-market a higher priority than reducing the number of faults in released software. In essence, it is more cost effe tive for companies to quickly release faulty software, then provide patches as needed, than it is to get the software right in the first place. Despite complaints about bugs from users, people have become accustomed to dealing with a certain number of faults in software; in fact, a certain number of faults are almost expected.

3. Limitations in the scope of formal methods prevent most complex software systems from being specified entirely using formal methods. Formal methods can only be applied when a problem can be modeled mathematically, but formally modeling even seemingly simple tasks such as sorting a collection of items can be overwhelming.

4. Formal methods are not scalable. As the size of a system increases, the amount of effort required to formally specify the system quickly becomes unsustainable.

One common thread in all these reasons is the sheer complexity of a typical software project. This is not to say that constructing a building, for example, is trivial, but the set of subproblems that must be solved in order to solve the main problem of constructing a building are all fairly well defined. Additionally, a building will not be built (usually, anyway) if there is some particular aspect of the project that has not been adequately solved. By contrast, many software projects are characterized by subproblems that are either not well defined, or are too difficult to define, based on incomplete understanding of the problem being solved, or on permutational or combinational complexity. Consider the game of chess, for example. Chess is limited to an 8x8 board, with a total of 32 pieces (16 pieces per player). There is a simple set of rules that govern movement of pieces, and that clearly define what constitutes the possible outcomes of a game. Chess is easy enough for most people to play even if they are not very good at it. But the problem of developing software capable of holding its own, or even defeating a world champion chess player was only solved fairly recently, and to date a complete solution for the game has yet to be determined, simply because of the enormous size of the search space.

The bottom line is, the amount of effort required for formal specification, coupled with the cost it incurs, makes it too much of a risk in the eyes of the software industry. One exception to this is in the area of critical systems, especially safety-critical systems, where a software failure could mean loss of life. The natures of those systems are deemed worthy of the effort required for formal specification. Since the use of formal methods should reduce the number of faults in the system, theoretically less effort should need to be spent on testing; this savings is balanced by the increased effort required during the requirements elicitation and design phases of development. The process model itself tends to resemble the waterfall model, where nothing is coded until it has been very thoroughly reviewed beforehand. In the past, some companies maintained so-called formal methods “gurus” on staff solely to help with creating formal specifications.

## Approaches to Formal Specification

There are two primary approaches used to formally specify a software system:

1. Algebraic, which focuses on describing how the various operations relate to each other

2. Model-based, which focuses on using sets and sequences to mathematically model the possible states of the system; operations are defined in terms of how they modify the system state.

Either approach can be done diagrammatically, but languages and notations have also been developed for both approaches. One of the most well-known languages, called simply “Z”, works with the model-based approach. None of the languages is ideal; for certain problems, succinct, descriptive text can do a better job of conveying the necessary rigor in defining system states or operations.

As mentioned previously, formal specification is time consuming and costly, whether a formal specification language is used or not. The examples Sommerville describes in chapter 27 should give you some insight into how difficult it is to truly, completely, formally specify a task. In a formal specification, there should be no ambiguities, every possible scenario should be addressed (as far as feasibility permits), and there should be no holes in the understanding of the problem. Earlier in the semester I mentioned the notion of being able to automatically generate a functioning application from the specification—essentially eliminating the coding phase of development. The specification referred to would need to be a formal one for a computer program to be able to read it and generate the code automatically.