Object-Oriented Real-Time Systems

Åsmund Tjora

Abstract

As the complexity of many real-time systems increases, the use of object-oriented design methods becomes more popular. Many of the methods used in object-oriented design can be a great improvement in the design of real-time systems. The use of a good real-time framework can improve the development speed, reuse level, and reliability as the framework provides classes that perform tasks that are common in many real-time systems.

1 Real-time systems

It is difficult to give a precise statement of what a real-time system is. Krishna & Shin [4] gives a loose definition:

"Any system where a timely response by the computer to external stimuli is vital is a real-time system"

The main point of this definition is that of *timely response*, meaning that there are deadlines that must be met.

Real-time systems vary widely in complexity and usage, from small embedded systems in household appliances to large factory automation systems.

As with other computer systems, the complexity of many real-time systems are increasing. The use of good methods for development of these systems are therefore important.

1.1 Hard and soft deadlines

The deadlines of the tasks in a real-time system can be divided into two groups, one group for absolute deadlines that must be met all the time, and one for deadlines that should be met under normal operation, but may be missed on some occations.

A *Hard Deadline* is a deadline that must be met if the system is to work properly. If the deadlines are not met, it may result in a system failure. It is important that the system is designed in a way that guarantees that all hard deadlines are met.

A *Soft Deadline* is a deadline that, if not met, will result in a degradation of system performance, but not a system failure. It is often a design goal that the system shall be able to meet a given percentage of the soft deadlines, and that all deadlines are met under normal circumstances.

1.2 Periodic and aperiodic tasks

A periodic task is a task that is run periodically. Most tasks in control loops are periodic. This group is often called time-driven tasks, as it is the passing of time that is the "event" that starts these tasks.

An aperiodic task is a task that are triggered by external events other than the passing of time. This is also called event-driven tasks.

A problem with aperiodic tasks is that it makes it difficult to perform a schedulability analysis and guarantee that all deadlines are met. If most of the tasks in the system have soft deadlines, this is usually not a real problem, as an "event overload" is the exception more than the rule. In systems with many hard deadlines, we have to make sure that there is an upper limit to the number of events happening in a given time interval.

1.3 Distributed systems

Large real-time systems tend to be distributed. This is often a natural consequence of the controlled process itself being physically distributed. Different parts of the process may also have different requirements with regards to the computer system used. A typical situation is having microcontrollers handling details in the control of instruments. These microcontrollers may be connected to a local unit handling more advanced tasks for a group of instruments. The local units may again be connected to a central unit that is handling plant-wide tasks (figure 1).

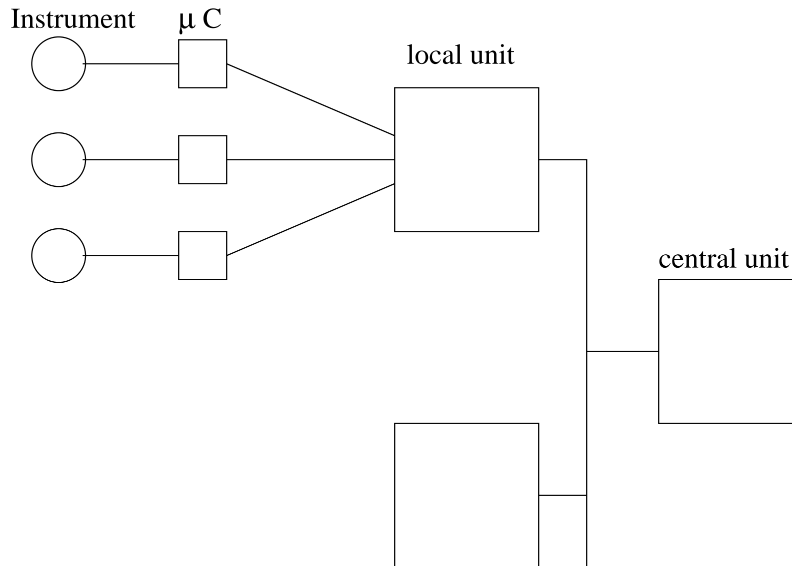

Figure 1: Example of a distributed system

Another reason to distribute the system is that the system load often cannot be handled by a single processing unit and still be able to meet the requirements, or that the cost of designing a system that can be run by a single processing unit may exceed the cost of a design that uses multiple units.

Distribution is also used for dependability purposes. By having more resources than actually needed, redundancy can be implemented in the system to increase the system reliability.

1.4 Scheduling

A real-time system usually has a lot of different tasks that are running concurrently. While this is also true for non-real-time systems, the requirements that deadlines must be met makes typical general purpose task schedulers like round-robin and FIFO unsuitable for most real-time systems.

The usual way to schedule real-time system tasks is to use a fixed priority, and always run the task with the highest priority that is ready to run. For a system with periodic tasks, the priority is usually set to be the inverse of the period (Rate Monotonic scheduling, RM) While the requirements to use the simple RM analysis (to decide if the system is schedulable) is not always present, the method usually gives a good idea on how priorities should be set.

For systems using a variable priority level, the Earliest Deadline First scheduling method is common: The task with the shortest time to deadline is given the highest priority.

1.5 The priority inversion problem

Another effect of having concurrent tasks is that several tasks may share resources. In a system there may be several *critical sections* that only one (or only a few) tasks may run at the same time. The usual way to hinder other tasks of entering the critical section is to use semaphores.

In real-time systems, the simple semaphore solution may cause a problem called *priority inversion*: Consider a low-priority task in a critical section. This task is preempted by a medium-priority task. While the medium-priority task runs, a high-priority task will be blocked from using the critical section, giving it an effective priority of the low-priority task.

The usual way to solve this problem is by using the *priority inheritance* protocol. When a low-priority tasks blocks a high-priority task, it will be given the same priority as the high-priority task, so it will run before medium-priority tasks. When it exits the critical section, it will return to its previous priority level.

Another method that may be used is the priority ceiling protocol. This method has much in common with priority inversion, but will also make sure that deadlocks cannot appear.

2 Using Object-Orientation in Real-Time Systems

As complexity of real-time systems increases it becomes necessary to use design models that can handle the system complexity while still making it possible for the developers to have a good overview of the system.

The advantages object-oriented design methods have when developing non-real-time applications is of course present when developing real-time systems. Douglass [1] lists the main advantages of using an object-oriented approach:

- Consistency of model views
- Improved problem-domain abstracton

- Improved stability in the presence of changes
- Improved model facilities for reuse
- Improved scaleability
- Better support for reliability and safety concerns
- Inherent support for concurrency

2.1 The control loop

Some of the views behind the use of object orientation fits well with many of the models behind the construction of a real-time system. As an example, real-time systems are often used to implement control systems, and a *block diagram* (figure 2) is often used to model the control system. The blocks in the diagram may be a part of the controller, the controlled system, or an instrument. The transitions between blocks in a detailed control loop model and objects in a real-time system design is not very difficult.

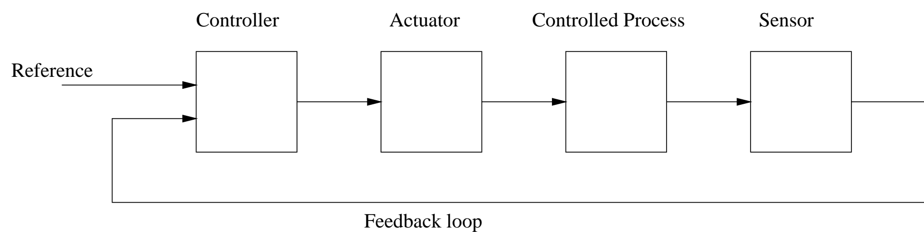

Figure 2: Simple control loop

2.1.1 Use of subclassing

Because many of the parts of the diagram is common in different types of systems, the possibility to reuse parts of a control system is good. As an example, the most common types of controllers are of the *PID* family, thus, once a PID controller is programmed, it can be reused for many different systems. By subclassing, features like anti-windup correction (useful in nonlinear systems where the control signal can be saturated) may be added to the controller. The PID controller itself may be implemented by adding integration and derivation to a simple propotional controller.

2.2 Roo

Another example of how subclassing can be used in real-time system can be found in the design of the *Roo* framework [8]. In Roo, both the scheduler mechanisms and the synchronization mechanisms has been developed by extending simple classes.

The basis of the scheduling class hierarchy is a simple round-robin scheduler (figure 3). By subclassing, priorities can be added, and by building on the new

PRIO class, the rate monotonic and earliest deadline first schedulers can be created. Another scheduler for real-time systems, *Least slack time first* has also been created by refining the EDF.

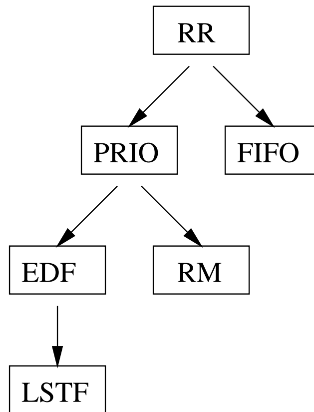

Figure 3: Inheritance graph for schedulers in Roo

Similarily, the synchronization mechanisms needed for real-time systems has been developed by using a base class implementing methods to block and release threads (figure 4), using this to create semaphore and monitor classes, and building the priority inheritance, priority ceiling and stack reservation protocols from the monitor class.

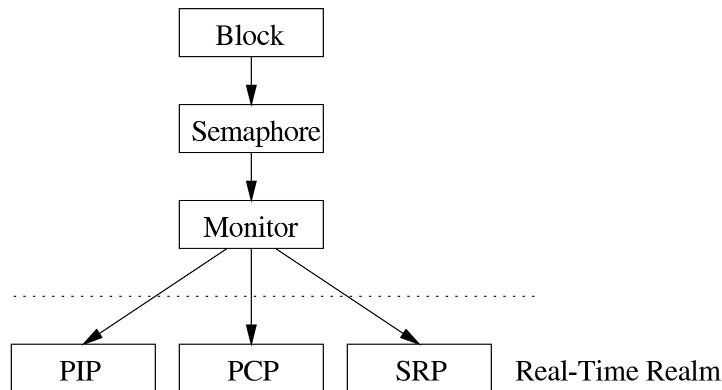

Figure 4: Inheritance graph for synchronization mechanisms in Roo

2.3 Physical representation

One of the main reasons to use a real-time system is that the system is interacting with the real world. It is often necessary to have a computer representation of the physical instrumentation (actuators and measuring instruments). The interface to the physical units can be modelled as objects where the physical state of the instrument is reflected as a logical state in the interface object.

Reading or writing these values is done by invoking methods on the interface objects.

2.4 Distribution of the system

As discussed in 1.3, real-time systems are often distributed. Some of the most popular design tools used for object-oriented design has built in methods for designing how the components of a system should be deployed on multiple processors. There has also been done some work on creating algorithms for allocation and scheduling object-based real-time systems on multiple processors [7].

2.5 Schedulability analysis

A problem with many of the current object-oriented modelling tools is the lack of tools for timing and schedulability analysis. This analysis is important in real-time systems, since it must be guaranteed that deadlines are met. Because many complex real-time systems are distributed, and because they usually have many event-driven tasks, many simple analysis tools are not usable.

3 Real-Time Frameworks

When developing an application, the developer may use a framework as a base for the application. "A framework is a partially completed application, pieces of which are customized by the user of the application." [1] The framework provides a common infrastructure for a set of applications

Frameworks may be divided into two groups:

- A *horisontally* oriented framework provides a deployment infrastructure that allows different applications operate in a common way or on a common platform.
- A *vertically* oriented framework is organized around a particular domain, and are typically more complete for applications in this domain.

3.1 Motivation

Douglass [1] lists a number of advantages by using frameworks:

- They provide a set of general ways to accomplish common programming tasks, which minimizes the complexity of the system, because there are fewer idioms to learn in order to understand the application structure and behavior.
- They provide service classes that perform many of the common housekeeping chores that make up most of all applications, which frees the developer to concentrate on domain-specific issues and problems.
- They provide a means to large-scale reuse for applications.
- They provide a common architectural infrastructure for your applications.

- They can greatly reduce the time-to-market for brand new applications
- They can be specialized for domains so that they can capture domain-specific idioms and patterns.

3.2 Patterns provided

A real-time framework is a vertical framework that is optimized for real-time applications. A real-time framework usually provides creational, structural, and behavioural patterns.

Examples of types patterns provided by real-time frameworks are:

- **Architectural support patterns** that represent strategic decisions that affect most of an application. Key decisions is the organization of domains and layers, and the principles of distribution across multiple processors and networks.
- **Collaboration and distribution patterns** like proxy and broker patterns are often provided as reliable solutions to common application domain problems.
- **Safety and reliability patterns** may be provided, but many of these patterns are often considered too application-specific to be used in a framework.
- **Behavioral patterns** like patterns that support the building of state machines.

3.3 What should be in a good framework

Douglass [1] gives a list of what a good framework should contain. A usable framework:

- Contains an appropriate set of services to be generally useful but not som many that a significant fraction is used by only a small set of applications.
- Has a reasonable generalization hierarchy that is neither too broad nor too deep
- Has service components that are plug-replaceable by similar components
- Is portable to different platforms and provides a consistent set of services to various applications
- Uses consistent naming and parameter-passing syntax
- Meets the performance and resource constraints of the applications within its domain

3.3.1 Set of services

A framework should be reusable in a variety of applications. This means that it does not require modification to be used. Instead the pieces of the framework is reused by subclassing parts of the framework, specialized or extending the the framework classes. Specializing a framework class means that the behavior polymorphic operations is altered in the subclass, so it suits the application better. Extending a class means that new attributes and methods are added to the subclass.

The services provided should be of general need by the applications in the target domain. If most applications don't use a service, it would be better to leave it out of the framework. Instead the developer should be able to use components from other libraries for the specialized services.

A real-time framework may include the following sets of services

- Task creation and management
- Resource management and synchronization
- Object distribution
- Object containment
- Object persistence
- Intra-processor communication
- State machine execution and event generation
- Event and interrupt handling
- Debugging facilites

3.3.2 Generalization Hierarchy Structure

There are two views on building class hierarchies. One is to provide a deeply nested hierarchy, so the framework contains many classes within the same taxonomy. The result of this is that the application developer usually only have to make small changes when using the classes. The other is to have greater steps in the specialization, creating a shallower hierarchy. This makes the framework smaller and more mantainable, but the application developer must make more changes when using the classes.

3.3.3 Replaceable components

As mentioned above, the framework should provide services that are of general need to the target domain. However, some applications will need specialized components not included in the framework. It is imporant that the framework allows for using a vareity of component libraries that are spezialiced for different needs.

3.3.4 Portability

Real-time systems are made for a variety of hardware and operating system platforms. Frameworks often make it easier move the application to a different execution platform by providing abstraction layers for hardware and operating system calls.

3.3.5 Performance

If the framework shall be usable for a real-time system, it must meet the performance requirements for the system. For real-time systems, the predictability of performance is vital, since it must be guaranteed that all deadlines are met under normal conditions and all hard deadlines must be met under any conditions. Since many real-time systems are embedded and have limited resources, it should be possible to exclude non-used parts of the framework.

4 Conclusion

Object-oriented design methods can be very useful when designing real-time systems. The use of objects and inheritance can make the design of a real-time system from the control models easier. The inherent support for concurrency and methods for distributing the system is also very important. There is a problem that many design tools does not have methods to perform timing and schedulability analysis.

Proper use of a good framework can speed up the development process. The framework will provide means for large-scale reuse of components, and will provide classes that perform common tasks and an architectural infrastructure.

References

[1] Bruce Powel Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* Addison-Wesley, 1999.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[3] Hermann Kopetz and Paulo Verissimo. Real time and dependability concepts. In Sape Mullender, editor, *Distributed Systems.* Addison-Wesley, 2nd edition, 1993.

[4] C. M. Krishna and Kang G. Shin. *Real-Time Systems.* McGraw-Hill, 1997.

[5] Gustaf Olsson and Gianguido Piani. *Computer Systems for Automation and Control.* Prentice Hall, 1992.

[6] Manas Saksena and Panagiota Karvelas. Designing for schedulability: Integrating schedulability analysis with object-oriented design. In *12th Euromicro Conference on Real-Time Systems*, pages 101–108, 2000.

[7] I. Santhoshkumar, G. Manimaran, and C. Siva Ram Murthy. A pre-run-time scheduling algorithm for object-based distributed real-time systems. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems*, pages 160–167, 1997.

[8] Chris Zimmermann and Vinny Cahill. Roo: A framework for real-time threads. In *Proceedings of the Third Workshop on Parallel and Distributed Real-Time Systems*, pages 137–146, 1995.