Team Programming for Real-Time Systems

Real-Time Innovations, Inc.

1 Introduction

Building software for complex real-time systems is a huge challenge. In addition to the problems facing any software development project, real-time software must coordinate actuators, sensors, and interactions with the real world. It must deal with complex and intelligent subsystems. It must be debugged “live” at run time, because the interaction with the system is crucial to the operation of the code. Many systems are large applications developed by teams of programmers, engineers, architects, and managers. Of course, safety and reliability are overriding concerns. Combined, these factors demand an efficient team development environment.

This team development requires specific tools for each member. This paper examines the requirements for building a successful large real-time software project. It:

- Explores approaches to system design, reuse, modularity, and testing;
- Analyzes the challenges in integrating powerful real-time software structures such as state machines and feedback systems, and
- Examines the approaches taken by several commercial tools.

A real-world application—a portion of the launch control system for the Space Shuttle—is used as an illustrating example.

2 The Real-Time Team Programming Challenge

Four challenges face teams writing complex software: coordination, communication, modularization, and reuse. This section examines each of these issues, with special emphasis on challenges facing real-time systems.

Coordination. Coordination of team talents and efforts is an issue in all software development. Real-time projects face an additional challenge: they often need to leverage many different areas of expertise. Talented programmers must work with a wide array of domain experts who may not be very good programmers, including hardware engineers, process experts, and test and integration specialists. Successfully developing an application that interfaces to a complex real-world device thus requires coordinating a diverse team with diverse skill sets.

Communication. As soon as a project grows beyond the capability of a single individual, communication becomes a critical problem for many reasons, including understanding the overall design, agreeing on interfaces, dividing the workload, and verifying that the design meets the desired specifications. All team members must both understand and contribute to the project.
There are many ways to communicate, including written, spoken, and graphical means. Every reasonably large project will require all of these.

Graphical diagrams are by far the most efficient of these methods; they communicate overall design much more effectively than any textual or spoken description. They are critical for overall system understanding, and must:

- Be commonly understood by all the team members;
- Intuitively fit the problem;
- Be sufficiently expressive to describe the problem;
- Communicate the design at the right level(s) of detail, and
- Map directly to the actual code.

These are stringent requirements, especially for the diverse real-time team. A notation that fits the object-oriented viewpoint of a software professional may seem hopelessly abstract to a controls engineer. Conversely, a control-systems feedback diagram may not capture the software structure of a complex system. Careful consideration of the problem is required before choosing a graphical notation; the notation must fit the specific problems faced by the team.

**Modularization.** Modularization, the process of dividing the task into functional units, is perhaps the most important factor in the success of a team project. A modular system is easier to design, build, and test. Done well, the modules can be developed in parallel, greatly accelerating the project.

However, modularization is nontrivial. Interdependencies, resource constraints, and functionality overlap must all be resolved before a useful division into modules can succeed. Most importantly, interfaces must be designed that determine what information flows between modules. Interface consensus—getting all sides to agree to well-defined interfaces—is the greatest challenge facing most projects.

Modularization is a well-known problem in the non-real-time programming world. Technologies such as CORBA and IDL strive to allow careful interface definitions. Unfortunately, these technologies are too heavyweight to help with the low-level efficient interfaces that real-time systems need. They do not deal with the timing and scheduling requirements of real-time systems. For effective programming and reuse, carefully-defined interfaces are required at all levels, from low-level functional blocks through high-level global objects.

**Reuse.** Finally, software reuse is a critical issue. Building systems from reusable software components is immensely powerful. Components speed development by leveraging already-written code. They are better-tested, higher-quality software. They can span teams, projects, companies, and even markets. The promise of component-based programming is huge.

The ability to reuse is largely an issue of structure. When the software environment is sufficiently structured, it is much easier to “plug in” separately-developed modules.

Reuse is also an issue of mentality. Without the acknowledged intent to reuse, the effort is unlikely to occur.

Finally, components must be easy to find, incorporate into new designs, test and maintain. Specific tools for building, browsing, and managing repositories of components improve greatly this process.
3 System Design Analysis

Coordination, communication, modularity and reuse are critical to team development. All four require structure. Choosing an appropriate structure requires close examination of the domain and its problems.

3.1 Structure

There are two ways to view a system: top-down and bottom-up. “Top-down” design consists of breaking the problem into smaller and smaller parts until the pieces can be implemented easily. Top-down design is very general and powerful and can eventually solve most problems. It is intuitively object oriented, it allows division of labor, and it encourages rapid prototyping through intermediate incomplete models of a complex system. It provides the “40,000 foot” viewpoint so important to communication and understanding between team members and management.

However, the top-down design process leads to unique solutions to every problem, requiring many custom-built modules that all have to be painstakingly hand crafted.

Bottom-up design, on the other hand, is the process of synthesizing from pre-existing components. Bottom-up design is great for reusing components. However, it often does not result in a clean vision or division of functionality. It provides no overall view to communicate the goals and progress of a project between team members.

A truly powerful solution requires a merged approach. Start with a global, undefined concept, and decompose it into more specialized subsystems. Along the way, define interfaces that modularize the design and insure clean divisions between subsystems. When the design is refined to the point that existing interfaces can be used, build the subsystems from the bottom up by combining components from a repository of reusable software. This merged approach combines the power of object modeling with the leverage of component-based synthesis.

This top-down / bottom-up combined approach is very intuitive. It allows designers to work simultaneously at different levels of abstraction. It results in a clean, object-oriented modular system built from reusable components.

However, it is not trivial to do. The two approaches are very different. They require fundamentally different thought processes. Merging them requires structure: a specific, well-defined object model and keen attention to module interfaces. There is a trade-off here; a very general tool cannot provide the structure required to make this work. You need a tool that supports the specific object structure that matches the problem. The challenge, therefore, becomes one of choosing a structure that matches your problem well.

3.2 Event and Cyclic Processing

One key to matching is to examine the types of processing your application requires. Most complex real-time systems have two very different fundamental requirements: cyclic data processing and event-driven reaction.

Consider a robot arm. At regular sample intervals, the controller reads the position and velocity sensors, computes where the arm should be, and outputs commands to the motors. All signal processing and control is thus done by a sampled-data system—code that is executed periodically at some set rate, such as 200 times per second.
Strategic motions are controlled by processing events and executing sequences of actions. The arrival of a command begins a sequence of steps that carry out its strategic objective. To pick up a moving object, the arm must compute an intercept trajectory, reach for the object, close the gripper, and then determine if the acquisition was successful. Each step in the sequence begins and terminates based on the arrival of an event.

Of course, different types of systems exhibit these properties to different extents. However, they are fundamental programming paradigms that drive the software structure.

3.3 Fit to Complexity

Another key to matching is to examine the sources of complexity in your application. Systems can be complex in many ways.

Some systems are complex because of the mathematical algorithms they contain. For example, a speech processing system has only one input (a microphone) and one output (text). The important challenge is developing a very complex mathematical algorithm.

Other systems are complex because they contain many concurrent elements. For instance, a telephone switch must make thousands of connections per second, and operate in an environment with many possibly distributed systems. Much of the challenge lies in understanding and modeling the interactions between these systems.

Other systems are faced with computational structure complexity. For instance, complex electromechanical systems must make strategic decisions, interface to intelligent modules and user interfaces, process data, execute feedback loops, change modes, and drive complex hardware. Interconnecting these subsystems requires complex interfaces, which leads to even more structural complexity. Many systems must also be modal, changing behavior in response to changes in operating conditions. This modal behavior can add very significant—and often unforeseen—structural complexity. The primary challenges are defining interfaces to allow modularization, and dealing with the intimate interactions between cyclic control and event processing.

Software architectures utilize structure to manage complexity. Since it is difficult to handle all types of complexity, architectures must make choices based on their target systems. Your challenge is to choose an architecture that mitigates the appropriate complexity of your system.

4 Approaches

It takes time to build a good real-time software structure. The required expertise is rare. The process is risky; many designs “run out of steam” only after months or even years of development.

With a proven structure, your risk is decreased. Your project will have more consistency. A structure allows libraries of generic components to be developed and used. It provides a known environment for tools. A proven structure that actually fits your problem is one of the greatest sources of leverage there is.

There are many commercial approaches. However, each approach is best suited to a particular type of problem. As we have seen, efficient team development cannot be achieved without a specific structure that fits the problem domain. In fact, many of the benefits of using a structure are realized only with a very specific structure; true power requires a very good match.

In this section, we look at the various approaches to real-time development and the problems they were designed to solve.
Mathematical Simulation Tools. There are several control and simulation tools on the market that allow you to build and test dynamic models of the system’s physical processes. They are designed for controls engineers who want to minimize coding, and are based on a cyclic processing paradigm. Even the state machine programming systems are based on a cyclic design; events are often limited to changes in numeric values, and are detected only at sampling intervals.

While these tools are excellent for mathematically complex systems, they are not designed to be good programming systems, and thus can’t handle systems that require significant custom coding or complex software structure.

General Programming Tools. There are many general-purpose programming tools on the market. Most are based on the Unified Modeling Language (UML). The UML is a very general and abstract analysis and modeling language, designed for use in applications ranging from databases to personnel management. Such general modeling languages let you describe arbitrary relationships between the various system objects. Each object can contain arbitrary functionality. However, the UML’s first primary function is “analysis”—the process of examining your problem and designing a structure that can solve the problem. This is a top-down approach; each problem leads to a unique structure. The UML tools thus require you to develop your own structure.

Telecommunications Tools. There are several architectures that concentrate on the telecommunications market. This problem domain is dominated by concurrency issues and event processing. Appropriately, these tools feature detailed message sequence analysis capability and concurrent state machine programming. While concurrency management is well-suited to telecommunications, it can inject artificial concurrency into applications where none is natural. As with the UML, neither cyclic nor numeric processing are addressed.

Structural Complexity Management. There are also frameworks designed for managing structural complexity. These tools focus on structure and interface definition. In the following example, we will examine an application of one of these tools, Real-Time Innovations’ ControlShell. ControlShell directly attacks computational structure complexity. It provides numeric and cyclic processing, but differs from the mathematical design tools because it is fundamentally designed as a programming system. It provides an object modeling system, but differs from the UML tools in that it provides a predefined structural framework targeted at a focused application.

5 Example

In this section, we take a more in-depth look at a real-world application: a portion of the Space Shuttle launch control system currently underway at NASA’s Kennedy Space Center (KSC).

5.1 Kennedy Space Center’s CLCS Program

The role the Checkout and Launch Control System (CLCS) program is to process and launch the Space Shuttle, automating (as much is possible) everything related to the Shuttle from the moment it lands to its launch.
The CLCS project is large and complex. The project will cost nearly $200M over its 5-year development cycle. It involves over 200 developers, plus engineers, test personnel, payload specialists, and more. Each launch is different, as the payload—usually a complex system in itself—is constantly changing. A large part of the automation involves sequencing sets of actions with appropriate error detection and recovery—a task well suited for ControlShell’s finite-state machine model. There are also many closed-loop feedback systems. Because processing the Shuttle involves an enormous amount of data (over 40,000 signals) and processing steps, managing the complexity is critical.

Forward Reaction Control System. The subsystem we will examine pressurizes the fuel and oxidizer tanks for a set of attitude control thrusters, the Forward Reaction Control System (FRCS). The pressurization is done by pumping high-pressure Helium gas into a bladder in each tank a few hours before launch. The pressurized Helium forces out the contents of the tank during flight.

As the tanks are pressurized, they become heated according to the ideal gas law, \( PV=nRT \), where \( P \) is the pressure, \( T \) is the temperature, and \( V, n \) and \( R \) are essentially constant. The heat is dissipated through contact with the environment. The goal is to pressurize both tanks so that they attain a desired pressure at ambient temperature without ever exceeding a safe temperature. The real system is much more complex than our treatment here; there are dozens of valves and sensors.

5.2 ControlShell

CLCS is using a toolset called ControlShell that specifically targets electromechanical systems. ControlShell systems are built from objects called Composite Object Groups, or COGs. COGs contain three types of components: event-driven state-machine subsystems, cyclic sampled-data subsystems, and reusable interface objects. COGs thus provide explicit, graphical integration of both fundamental types of processing and the interfaces required for modularity. COGs are fully hierarchical—you design complex systems by building COGs made up of other COGs.

ControlShell makes a bold claim: most electromechanical systems can be modeled as systems of COGs. Thus, ControlShell imposes a specific object model on the design. This specific model provides the structure needed to allow component-based programming. It also shortcuts much of the tedious analysis and object-hierarchy design phase.

ControlShell’s overall methodology is diagrammed in Figure 2. On the left, a control system is generated to control the tank filling process. Similarly, a state machine is built to control the pressurization sequencing. These functional subsystems are combined with an interface into the Tank COG in the center of the figure. This COG is combined with others to build the FRCS system, which in turn is part of the reaction control system.
The left side of the diagram is procedural programming. The right is object oriented. Working from left to right is bottom-up design. Working the other way is top-down. The tool set supports both equally. In practice, the design process is an iterative application of both approaches.

ControlShell is designed for team programming. It combines an engineering design tool with an open software design and programming system. The graphical diagrams are straightforward, designed to provide a common language that all team members can use and understand. Specific tools are provided for interface design, software development, system configuration, and test. Special emphasis is placed on enabling the domain engineers to control the design. ControlShell also provides a complete run-time environment, including dynamic simulation, extensive run-time libraries, and interactive debugging. It targets complex engineering systems that require significant custom code.

### 5.3 FRCS

The Forward Reaction Control System subsystem consists of two tanks (one for Fuel and one for Oxidizer), a regulator, and a controlling sequencer shown in Figure 3. Each tank has one valve that controls the helium intake. The regulator controls the pressure of helium to be applied to each tank.

**Repository.** ControlShell is built around an integrated component repository (on the left in Figure 3). Virtually everything is placed in the repository for potential reuse. Developers write new components and store them in repositories. Engineers then pull together systems built from components in the repositories. The repository stores components at many different levels of granularity, including method definitions, atomic code modules, sampled-data systems, state
machines, action routines, and COGs. Even the final subsystems are stored in repositories for use in prototyping, testing, implementation and maintenance. This repository-centric view of the world provides a reuse mindset that pervades the entire project.

**Interfaces.** ControlShell provides explicit tools to define interfaces. Once defined, an interface becomes a reusable object that is stored in the repository as any other component. Interfaces can be included in COGs, attached to most any object, exported to higher-level objects, or connected internally.

The tank interface shown in Figure 3 contains both sensed data (temperature and pressure), and methods (to open and close the valve). There are two instances of the interface. The Oxidizer tank interface in the figure is opened to show its contents (top right).

**The Plant Model.** A simplified plant model for the simulated tank system is presented in the sampled-data diagram in Figure 4. The diagram models the tank valve (top left), and the tank mass and pressure (top row). The interface (top right) connects the sequencer to this COG. The block in the lower left (HeliumTempModel) models the heat exchange between the pressurized Helium gas and the tank, the TankTempModel block models the exchange of heat between the tank and the external environment. These heat exchange modules are custom-written code for
Pressurization Logic. The top-level controlling sequencer is shown in the state-machine diagram in Figure 5. The system starts out in the Idle state. When the command to pressurize the tanks is given (“Command = Run”), the system enters the “Running” level. If the command is ever changed to “Secure” (indicating an emergency shutdown), the pressurization is aborted by the Securing routine. Hierarchical stimulus processing makes it simple to handle this exception condition. After the initial status checks and regulator verification phases, the system enters the pressurization composite substate shown in Figure 6.

The subroutine reuse capability is handy in Figure 6, allowing multiple calls to the individual tank pressurization routines. The (simplified) PressurizeTank pressurization subroutine is shown in Figure 7. This routine opens the set of valves required to pressurize the tank. It then cycles between pressurizing and cooling until the monitors indicate the pressure is as desired. The system then waits ten minutes in the Stabilizing state to insure the pressure and temperature are both stable.

To support the COG model, the state machine diagrams support data connections as well as states. Each transition is represented by a visual element that can be hooked up to methods and data. Substates are also represented by an icon that presents all the exported data and method pins. For instance, the PressurizeTank subroutine monitors the pressure and temperature values, and calls the OpenValve and CloseValve methods.
Execution. The same diagrams can drive the simulation or run the actual hardware. Figure 8 shows the initial pressure and temperature cycles, as displayed by the StethoScope run-time monitor. The pressurization curve for the entire cycle is shown in Figure 9.

Figure 9 Pressurization Progress
The pressure rises at constant temperature. The model provides an accurate estimate of the final desired pressure, and stops the pressurization so the cooled system is at the target pressure.

On line testing tools are critical during both system development and operation.

Of course, the FRCS is just part of a much larger system. However, the structure shown here—a hierarchy of COGs with well-defined interfaces—is used to model and control the entire system.
Figure 6 Pressurization Choice Subroutine

This is one of many subroutines in the system. It determines which tanks will be pressurized. The blocks on the right allow you to hookup data and methods in sampled-data systems.

Figure 8 Simulation Start-up

When the valve opens (lower blue line), the temperatures of both Helium and Oxidizer rise. The valve cycles when the temperature reaches the safe limits.
Figure 7 Tank Pressurization Subroutine

This is the lowest-level sequencing. It cycles between pressurizing and cooling the tank. The oxidizer and fuel tanks each contain one of these routines. Each state transition routine has a corresponding visual icon (on the left) that allows you to hook up data and methods to external COGs.

6 Conclusions

Efficient team development requires structure. It is precisely the existence of common structure that allows the team to work together, communicate ideas and designs, divide the work, and build from reusable components. Structure enables component-based development without sacrificing top-down design. A software architecture and toolset that match your problem is an amazingly powerful resource.

However, the structure must match your problem well. The source of complexity, processing requirements, and team makeup determine the quality of the match. Consider each carefully, then choose the tool that provides the most specific support possible but is still sufficiently general to solve your problem. With the right structure and the right toolset, your team can solve even the most complex real-time problems on time and under budget.