AIAA Conference on Intelligent Robots in Field, Factory, Service, and Space, March 1994

ControlShell: A Real-Time Software Framework

Stanley A. Schneider Vincent W. Chen Gerardo Pardo-Castellote

Real-Time Innovations, Inc.
954 Aster
Sunnyvale, California 94086

Aerospace Robotics Laboratory
Stanford University
Stanford, California 94305

Abstract

This paper describes ControlShell, *a next-generation CASE "framework" for real-time system software development. ControlShell's well-defined structure, graphical tools, and data management provide a unique component-based approach to real-time software generation and management. ControlShell is designed specifically to enable modular design and implementation of real-time software. By defining a set of interface specifications for inter-module interaction, ControlShell provides a basis for real-time code development and exchange.*

ControlShell includes many system-building tools, including a graphical data flow editor, a component data requirement editor, and a state-machine editor. It also includes a distributed data flow package, an execution configuration manager, a matrix package, and an object database and dynamic binding facility. ControlShell is being used in several applications, including the control of free-flying robots, underwater autonomous vehicles, and cooperating-arm robotic systems.

This paper presents an overview of the ControlShell architecture, and details the functions of several of the tools.

1 Introduction

Motivation System programs for real-time command and control are, for the most part, custom software. Emerging operating systems [1, 2, 3, 4, 5] provide some basic building blocks—scheduling, communication, etc.—but do not encourage or enable any structure on the application software. Information binding and flow control, event responses, sampled-data interfaces, network connectivity, user interfaces, etc. are all left to the programmer. As a result, each real-time system rapidly becomes a custom software implementation. With so many unique interfaces, even simple modules cannot be shared or reused.

An effective real-time framework must create a programming environment that facilitates sharing and reuse of real-time program modules. At a minimum, this requires providing interface specifications and data transfer mechanisms. The framework must also provide services and tools to combine modules and build systems from reusable components. Finally, the framework must meet the many challenges unique to real-time computing. For example:

- Real-time code must be able to react to external temporal events.
- The real-time execution environment is *fundamentally* multi-threaded and asynchronous.
- Real-time systems are usually composed of several different layers of control, each with different characteristics. For instance, strategic-level command and low-level servo control must be blended into a smoothly-operating system.
- Real-time systems must handle changing conditions, often requiring switching between drastically different modes of operation.
- Real-time systems are often physically distributed. In the simplest case, an operator control station may be remotely situated. More complex systems are comprised of many interacting distributed real-time and non-real-time subsystems.

All these challenges must be efficiently and smoothly handled by the architecture.

1.1 ControlShell's Solutions

Component-Based Design ControlShell is specifically designed to address these issues. ControlShell provides interface definitions and mechanisms for building real-time code modules. ControlShell also provides basic data structure specifications, and mechanisms for binding data with routines and specifying data-flow requirements. These two critical features make simple generic packages (known as *components*) possible. ControlShell systems are built from combinations of these components.

An extensive library of pre-defined components is provided with the system, ranging from simple filters and controllers to complex trajectory generators and motion planning modules. New or custom components are easily added to the system via the graphical *Component Editor* (CE). The Component Editor allows simple specification of data interchange requirements. Code is automatically generated to permit instancing the new component into the system.

Graphical CASE System-Building Tools ControlShell also provides a set of powerful development tools for building complex systems. Building a system is accomplished by connecting components within a graphical *Data Flow Editor* (DFE). The data flow editor resolves the system data dependencies and orders the component modules for most efficient execution. Radical mode changes are supported via a "configuration manager" that permits quick reconfiguration of large numbers of active component routines.

Real-time systems also require higher-level control functions. ControlShell's event-driven finite state machine (FSM) capability provides easy strategic control. The state machine model features rule-based transition conditions, true callable sub-chain hierarchies, task synchronization and event management. A graphical FSM editor facilitates building state programs.

Real-Time System Services To provide support for real-time distributed systems, ControlShell includes a network connectivity package known as the *Network Data Delivery Service* (NDDS). NDDS provides distributed data flow. It naturally supports multiple anonymous data consumers and producers, arbitrary data types, and on-line reconfiguration and error recovery.

ControlShell also offers a database facility, direct support for sampled-data systems, a full matrix package, and an interactive menu system. Figure 1 presents an overview of the ControlShell toolset and design approach.

2 Relation to Other Research

There are two quite different issues in real-time software system design:

- Hierarchy (what is communicated)
- Superstructure architecture (how it is communicated)

Several efforts are underway to define hierarchy specifications; NASREM is a notable example [6]. ControlShell makes no attempt to define hierarchical interfaces, but rather strives to provide a sufficiently generic software platform to allow the exploration of these issues. As such, this work takes a first step—defining the architecture superstructure (control and data flow mechanisms).

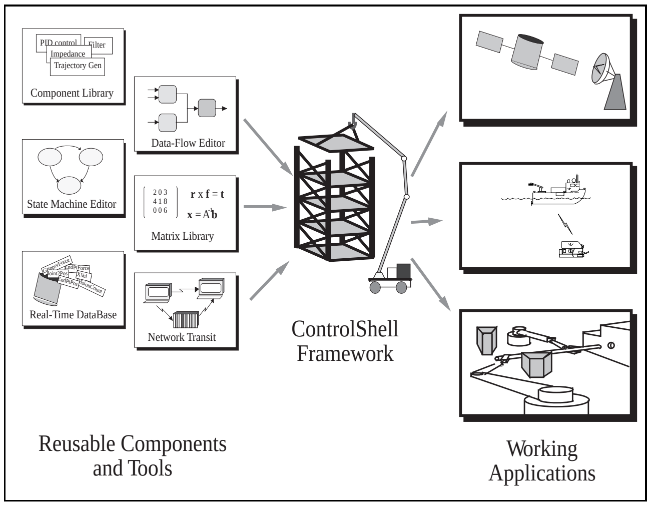


Figure 1: **ControlShell Design Process**

The ControlShell system designer uses the many powerful tools, system services, and prebuilt library components to construct a modular system.

Several distributed data-flow architectures have been developed, including CMU's TCA/X [7, 8], Rice University's TelRIP [9], and Sparta's ARTSE [10]. These provide various levels of network services, but do not address repetitive service issues or resolve multiple data-producer conflicts in a symmetric robust "stateless" architecture as does the ControlShell NDDS system (see [11] for details). Also, they are not integrated within a general programming system.

Recently, more sophisticated programming environments have begun to emerge. For example, ORCAD [12] and COTS [13] are specialized robotics programming environments. Two commercial products, System Build with AutoCode from Integrated Systems, Inc. [14], and SIMULINK with C-Code Generation from the MathWorks, Inc. [15] are sophisticated control development environments. They offer easy-to-use rapid control system prototyping. They are not, however, architectures well suited to developing complex multi-layer distributed control hierarchies.

Implementation Experience ControlShell evolved from many years of research with real-time control systems. It was first developed for use with a multiple-arm cooperative robot project at Stanford University's Aerospace Robotics Laboratory[16, 17, 18]. From this start, ControlShell spread to become the basis for more than 20 research projects in advanced control systems at Stanford. Among these were projects to study the control of flexible structures, adaptive control, control of mobile robots (including multiple coordinated robots), and high-bandwidth force control [19, 20, 21, 22, 23, 24].

More recently, a few industrial sites and two NASA centers have begun experimenting with ControlShell applications [25, 26]. ControlShell is now being jointly developed by Stanford University and Real-Time Innovations, Inc. It is supported under ARPA's Domain-Specific Software Architectures (DSSA) program.

This *continuous migration* from specific, working applications to wider spectrums of use is the key to usable generality. These applications continue to drive ControlShell's growth. To our knowledge, ControlShell is the only integrated framework package combining transparent networking, component-based system description, a state machine model, and a run-time executive.

3 Run-Time Structure

Some of the major system modules are shown in Figure 2. As shown in the figure, ControlShell is an *open* system, with application-accessible interfaces at each level. The figure is organized (loosely) into data and execution hierarchies.

At the lowest layer, ControlShell executes within the VxWorks real-time operating system environment. The simple base class known as *CSModules* is the building block for most executable constructs. Organizations of these modules, into lists, menus, and finite state machines form the core executable constructs. Users build useful execution-level atomic objects called *components* by defining derived classes from CSModules and binding them through the on-line data base to data matrices from the *CSMat* package. High-level graphical editors speed component definition, data flow specification and state machine programming. Network connectivity is provided by NDDS for all application modules.

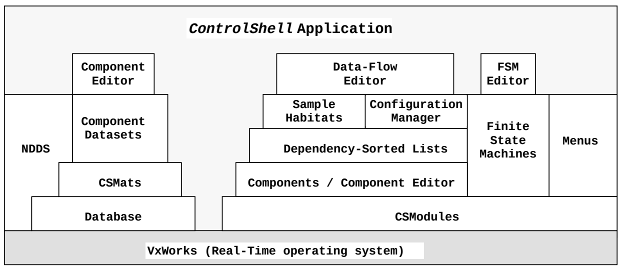


Figure 2: **Run-Time Structure**

ControlShell's open architecture provides many powerful services, while allowing application code free access to the underlying structures.

4 Data Flow Design

Many real-time systems contain sampled-data subsystems. Here, we define a "sampled-data" system as any system with a clearly periodic nature. Common examples (each of which have been implemented under ControlShell) are digital control systems, real-time video image processing systems, and data acquisition systems. Each of these is characterized by a regular clock source.

Providing an environment where sampled-data program components can be interchanged is challenging. These programs have routines that must be executed during the sampling process, routines to initialize data structures (or hardware) when sampling begins, and perhaps to clean up when sampling ends. Further, many routines are dependent on knowledge of the timing parameters, etc. Although they may interact—say by passing data—sampled-data program components are often relatively independent. Requiring the application code to call each module's various routines directly destroys modularity.

4.1 Components

The *component* is the fundamental unit of reusable data-flow code in ControlShell. Components consist of one or more *sample modules* derived from CSModules. Sample modules have several pre-defined entry routines, including:

Routine	**When executed**
execute	Once each sample period
stateUpdate	After all executes are done
enable	When this module is made active
disable	When it is removed from the active list
startup	When sampling begins
shutdown	When sampling ends
timingChanged	When the sample rate changes
reset	When the user types "reset", or calls CSSampleReset
terminate	When the module is unloaded

Thus, a motor driver component might define a startup routine to initialize the hardware, an execute routine to control the motor, and a shutdown routine to disable the motors if sampling is interrupted for any reason. In addition, if any of its parameters depend on the sampling rate, it may request notification via a timingChanged method. By allowing components to attach easily to these critical times in the system, ControlShell defines an interface sufficient for installing (and therefore sharing) generic sampled-data programs.

Building Components: The Component Editor An easy-to-use graphical tool called the *component editor* (CE) assists the user in generating new components and specifying their data-flow interactions. The component editor defines all the input and output data requirements for the component, and creates a data type for the system to use when interacting with the component. The tool contains a code generator; it automatically generates a description of the component that the Data-Flow Editor can display (see below), and the code required to install instances of the component into ControlShell's run-time environment.

4.2 Execution Lists

An execution list is simply a dynamically changeable, ordered list of sample modules to be sequentially executed. The active set of modules on a list can be changed anytime. In fact, lists may drastically change their contents during system mode changes.

Execution lists may be sorted to provide automated run-time execution scheduling to resolve data dependencies. More specifically, the modules are sorted so that data consumers are always preceded by the appropriate data producers (see Figure 3). The system uses the specifications of the data flow requirements for each component to sort the dependencies and order the list. A side benefit of the sorting process is the error-checking that is performed to insure consistent data flow patterns.

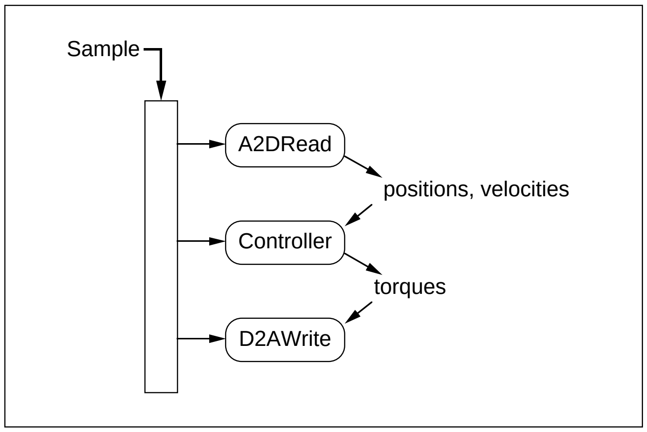


Figure 3: **Dependency-Sorted List**

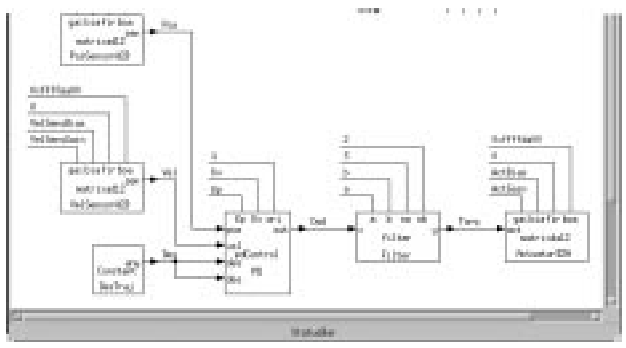
Dependency-sorted execution lists provide automatic run-time sorting by data *dependencies.*

Sample Habitats ControlShell provides a named sampled-data environment, known as a *sample habitat*. A sample habitat encapsulates all the information and defines all the interfaces required for sampled-data programs to co-exist. It also contains routines to control the sampling process. For example, a module installed into a sample habitat can query its clock source and sample rate, start and stop the sampling process, etc.

Each sample habitat contains an independent task that executes the sample code. The task is clocked by the periodic source (such as a timer interrupt). Special components are provided to interface between habitats, allowing multi-rate controller designs.

4.3 Building Systems: The DFE Editor

Building systems of components is made simple by the graphical Data-Flow Editor (DFE). The DFE reads description files produced by the component editor, and then allows the user to connect components in a friendly graphical environment. It allows specification of all the data connections in the system, as well as reference inputs—gains, configuration constants and other parameters to the individual components. An example session is depicted in Figure 4.


pdControl
PD
Filter
Filter