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.

Copyright © 1993 by Real-Time Innovations, Inc. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission.
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).

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.

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:

<table>
<thead>
<tr>
<th>Routine</th>
<th>When executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute</td>
<td>Once each sample period</td>
</tr>
<tr>
<td>stateUpdate</td>
<td>After all executes are done</td>
</tr>
<tr>
<td>enable</td>
<td>When this module is made active</td>
</tr>
<tr>
<td>disable</td>
<td>When it is removed from the active list</td>
</tr>
<tr>
<td>startup</td>
<td>When sampling begins</td>
</tr>
<tr>
<td>shutdown</td>
<td>When sampling ends</td>
</tr>
<tr>
<td>timingChanged</td>
<td>When the sample rate changes</td>
</tr>
<tr>
<td>reset</td>
<td>When the user types &quot;reset&quot;, or calls CSSampleReset</td>
</tr>
<tr>
<td>terminate</td>
<td>When the module is unloaded</td>
</tr>
</tbody>
</table>

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.

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.

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.

Figure 3: Dependency-Sorted List
Dependency-sorted execution lists provide automatic run-time sorting by data dependencies.

Figure 4: Data-Flow Editor
The data-flow editor builds collections of components into an executing system.

5 Configuration Management
Complex real-time systems often have to operate under many different conditions. The changing sets of conditions may require drastic changes in execution patterns. For example, a robotic system coming into contact with a hard surface may have to switch in a force control algorithm, along with its attendant sensor set, estimators, trajectory control routines, etc.

ControlShell's configuration manager directly supports this type of radical behavior change; it allows entire groups of modules to be quickly exchanged. Thus,
different system personalities can be easily interchanged during execution. This is a great boon during development, when an application programmer may wish, for example, to quickly compare controllers (See Figure 5). It is also of great utility in producing a multi-mode system design. By activating these changes from the state-machine facility (see below), the system is able to handle easily external events that cause major changes in system behavior.

Configuration Hierarchy The configuration manager essentially creates a four-level hierarchy of module groupings. Individual sample modules form the lowest level. These usually implement a single well-defined function. Sets of modules, called module groups, combine the simple functions implemented by single modules into complete executable subsystems.

Each module group is assigned to a category. One group in each installed category is said to be active, meaning its modules will be executed. Finally, a configuration is simply a specification of which group is active in each category.

Figure 5: Configuration Manager
Configurations can be swapped in or out under program or menu control. This provides flexible run-time reconfiguration of the execution structure.

Example As a simple example, consider a system with two controllers: a proportional-plus-derivative controller named “PD”, and an optimal controller known as “LQG”. Suppose the PD controller requires filtered inputs, and thus consists of two sample modules: an instance of the PDControl component and a filter component. These two components would comprise the “PD” module group. The “LQG” controller module group may also be made up of several components. Both of these groups would be assigned to the category “controllers”.

The user (or application code) can then easily switch controllers by changing the active module group in the “controller” category.

Now suppose further that the controllers require a more sophisticated sensor set. A category named “sensors” may also be defined, perhaps with module groups named “endpoint” and “joint”. The highest level of the hierarchy allows the user to select an active group from each category, and name these selections as a configuration. Thus, the “JointPD” configuration might consist of the “joint” sensors and the “PD” controller. The “endptLQG” configuration could be the “endpoint” sensors and the “LQG” controller.

Category and Group Specification This subdivision may seem complex in these simple cases. However, it is quite powerful in more realistic systems. It has been shown to be quite natural in applications ranging from a vision-guided dual-arm robotic system able to catch moving objects [16] to flexible-beam adaptive controllers [27].

Assigning modules to groups and groups to categories is made quite simple with the ControlShell graphical DFE editor’s “configuration definition” window, shown in Figure 6. New categories are added with the click of a button. To create a module group, the user simply names a group, and then clicks on the modules in the data-flow diagram that should belong to that group. The blocks are color-coded to relate the selections back to the user.

Figure 6: Configuration Definition
Configurations are easily defined within the DFE graphical interface.
6 Finite State Machines

A real-time system in the real world must operate in a complex, event-driven environment. With only a sequential programming language, the burden of managing and reacting to events is left to the programmer.

The Finite State Machine (FSM) module is designed to provide a simple strategic-level programming structure that also assists in managing events and concurrency in the system. The FSM module combines a non-sequential programming environment with natural event-driven process management. With this structure, the programmer is actively encouraged to divide the problem into small, independently executing processes.

To utilize the FSM module, the programmer first describes the task as a state transition graph. The graph can be directly described within ControlShell's graphical FSM editor (see Figure 7). Each transition—represented by an arrow in the graph—specifies a starting state, a boolean relation between stimuli that causes the transition, the CSModule to be executed when the transition occurs, and a series of "return code-next state" pairs that determine the program flow.

The FSM model is quite general; it supports rule-based transition conditions (reducing the number of states in complex systems), true callable sub-chains of states (so libraries of state subroutines can be developed), wild-card matching (so unexpected stimuli can be processed), global matching (allowing easy error processing), and conditional succession (so state programs may easily branch). Transitions are specified as boolean relations of three types of stimuli: transient, latched, and conditional. Transient stimuli have no value, and exist only instantaneously. Latched stimuli also have no value, but persist until some transition expression matches. Condition stimuli have string values; they persist indefinitely and thus represent memory in the system. Thus, the transition condition "Object = Visible AND Acquire" might cause a system to react to an acquisition command from a high-level controller. Providing these three stimuli types allows combination of both "system status" and "event" types of asynchronous inputs into easily-understood programs.

The FSM module takes advantage of the atomic message-passing capability of modern real-time kernels to weave the incoming asynchronous events into a single event stream. Any process can call a simple routine to queue the event; the FSM code spawns a process to execute the resulting event stream. The result is an easy-to-use, yet powerful real-time programming paradigm.

7 Data Control and Binding

Most data in a ControlShell application is embodied in CSMats. A CSMat is a named matrix of floating-point values. Each row and column of the matrix operationally contains a field name and a units specification. A complete real-time matrix mathematics utilities package is included. Components may combine multiple CSMats into structures for efficient reference and parameter passing.

The entire control hierarchies are created and bound to the correct data objects at run-time. The system is built from the graphically-generated description files produced by the DFE and FSM editors. This dynamic binding paradigm is very powerful—it combines the convenience of automatic system building with the flexibility of a dynamically changeable system. Thus, it provides the features of a full code generation without the pre-compiled inflexibility.

To support this dynamic binding, ControlShell incorporates a "linking" database facility. All instances of each data object (such as CSMats)—and each control construct (such as execution lists)—are entered into the database upon creation. The database allows "reference before creation" semantics for many object types; if a requested object is not in the database (i.e. it does not exist), an incomplete (e.g. zero-sized) object will be created by the database itself. This capability allows considerable flexibility at run-time; modules may, for instance, specify dependencies on data sets that do not yet exist, etc. Verification routines insure that the system is consistent before actual "live" execution begins.

8 Network Connectivity

ControlShell is integrated with a network connectivity package called the Network Data Delivery Service.
NDDS [11]. NDDS is a novel network-transparent data-sharing system. NDDS features the ability to handle multiple producers, consumer update guarantees, notifications or "query" updates, dynamic binding of producers and consumers, user-defined data types, and more.

The NDDS system builds on the model of information producers (sources) and consumers (sinks). Producers register a set of data instances that they will produce and then "produce" the data at their own discretion. Consumers "subscribe" to updates of any data instances they require. Producers are unaware of prospective consumers; consumers are not concerned with who is producing the data they use. Thus, the network configuration can be easily changed as required. NDDS is a symmetric system, with no "special" or "privileged" nodes or name servers. All nodes are functionally identical and maintain their own databases. The routing protocol is connectionless and "quasi-stateless"; all data producer and consumer information is dynamically maintained. Thus dropped packets, node failures, reconfigurations, over-rides, etc. are all handled naturally.

This scheme is particularly effective for systems (such as distributed control systems) where information is of a repetitive nature. NDDS is an efficient, easy-to-use distributed data-sharing system. Figure 8 illustrates the use of NDDS within a cooperating-arms robot system (see [24]).

Figure 8: Network Data Delivery Service

NDDS provides a network "backplane". Each module can easily share data with any other module. The individual connections are handled transparently by the system.

9 Conclusions

This paper has presented a brief overview of the capabilities of the ControlShell system. ControlShell is designed—first and foremost—to be an environment that enables the development of complex real-time systems. Emphasis, therefore, has been placed on a clean and open system structure, powerful system-building tools, and inter-project code sharing and reuse.

Acknowledgements

ControlShell is being jointly developed by Stanford University and Real-Time Innovations, Inc. It is currently supported under ARPA's Domain-Specific Software Architectures (DSSA) program. The authors wish to express their gratitude to Dr. Marc Ullman for his many contributions to this project, and Dr. R. H. Cannon, Jr. for his guidance and leadership. The authors would also like to thank the many developers at Stanford, Loral, and NASA who have contributed ControlShell components.

References


