1 Introduction

1.1 Abstract

Real-time system software is notoriously complex. Large projects must develop complex software and hardware with interacting teams of programmers, engineers, managers, and maintenance personnel. To succeed, projects need a solid software architecture, an intuitive development environment, a way to leverage reused code, and practical tools that really accelerate the work.

ControlShell, the revolutionary graphical programming tool for intelligent control, provides all of these. ControlShell goes well beyond object-oriented analysis; it provides a proven application framework that ensures quality software architecture. Its component-based design includes an integrated code repository and powerful interface definition tools, making reuse practical. ControlShell is a true programming system, sophisticated enough for talented programmers building truly complex systems, but straightforward enough not to lose you in abstractions. It simply gets complex systems running sooner.

ControlShell is ideal for team development. Designed for real-time engineers by real-time engineers, ControlShell lets talented programmers and engineers work together. Both programmers and engineers will find ControlShell merges naturally into their current development process.

ControlShell is the result of many years of research, both at RTI and Stanford University. It’s a truly new way of building software. It has been applied to many complex systems, ranging literally from deep-sea vehicles to interplanetary robotics. It is now finding application in large-scale mission-critical applications such as launching the Space Shuttle. Take a tour of ControlShell with us in these pages. We will point out how and why ControlShell is the unique solution you need. We’ll try not to bore you with platitudes.

But first, let us review our three main points: ControlShell is the right solution for intelligent control, it fits naturally into your process, and it speeds your development.

1.1.1 ControlShell is the Right Solution for Intelligent Control

ControlShell specifically targets intelligent control and simulation systems. It combines modern component-based programming with real-time control-system mathematics. ControlShell is more focused on control systems than abstract software modeling tools, so it provides the environment you need to get your controllers running quickly. ControlShell is more focused on software structure than mathematical simulation tools, so it can handle the complex, custom code your application needs.

ControlShell scales to handle complex systems. It makes hierarchy natural by carefully specifying interfaces, and then binding interfaces to objects that combine dataflow (control system) diagrams with event-driven state machines. It builds complex systems from nested diagrams made of these reusable components. This architecture couples object-oriented software design with engineering diagramming. It’s an intuitive, general approach that matches how engineers and developers naturally think.
Finally, ControlShell lets you leverage the scalable power of component-based programming. It easily incorporates legacy and custom code. It handles complex systems that change modes of operation with a unique hierarchical mode-mapping technique. And ControlShell’s scalability is proven; successful applications range from small, embedded systems to the huge launch control system for the Space Shuttle.

1.1.2 ControlShell Fits Naturally Into Your Team’s Process

ControlShell recognizes that engineering teams are diverse, and provides specific tools for each phase of the project. ControlShell leverages everyone’s skills, including system architects, programmers, domain engineers, and test and integration specialists. Its graphical language provides the critical communications medium that ties the team together.

ControlShell’s intuitive engineering diagrams put domain experts in control of the project. It lets engineers build systems from pre-defined blocks, leveraging code written by others. It supports real-time matrix mathematics, making it straightforward to build complex algorithms. It directly supports hardware-in-the-loop simulation by letting you “mix and match” real hardware with simulated modules. It’s tightly integrated with Simulink, so you can import your detailed control designs. ControlShell gives engineers the tools they need to understand and drive the project.

At the same time, ControlShell won’t frustrate good programmers. ControlShell users create a clean object-oriented design from the “top-down” by decomposing the problem into interacting objects and interfaces. You then build your system from the “bottom-up” with components chosen from open-source repositories. This open, integrated approach excels at complex software designs.
ControlShell fits into and enhances your current system development process. Without changing your teams, ControlShell lets everyone use the same tool. Control designers, programmers, system engineers and test groups can share designs, code, and documentation. Transitions between groups are automated. ControlShell streamlines your entire software development process.

1.1.3 ControlShell Speeds Your Development

ControlShell accelerates your work. ControlShell’s fully-automated visual development is fast and easy to use. It automatically generates component code, builds the system, and links it on the target. All capabilities are available from a single tool, including diagram editing, source-code editing, code generation, and automatic HTML documentation. It features a unique “on-target” build system that eliminates most compiling and makes the analyze/edit/test cycle blazingly fast. It integrates run-time tools for test scripting, code debugging, data capture and visualization, state machine testing, execution profiling, and memory analysis. ControlShell teams deliver a higher quality, more reliable, more maintainable, and better-documented system for less money in less time. (OK, so we threw in one platitude.)

1.2 Focus

The Challenge. Building software for complex real-time systems is a huge challenge. The software must coordinate actuators, sensors, and interactions with the real world. It must deal with complex and intelligent subsystems. Many systems are large applications developed by teams of programmers, engineers, architects, and managers. All team members must be able to both understand and contribute to the design.

The demands for a modular, clean design are compounded by the special needs of complex real-time systems, including control, sequencing, and on-line debugging. The system must be built, simulated and tested incrementally. Significant legacy code embodying years of experience must be employed. In the past, these systems defied reorganization. But now, the marketplace demands better, faster, cheaper systems. It’s time for a new approach.

Target Applications. ControlShell targets intelligent control systems. By “intelligent”, we do not imply Artificial Intelligence techniques as defined by computer scientists, nor the fuzzy, adaptive, or genetic algorithms popular in the research controls community. While an intelligent control system may incorporate an AI module or fuzzy controller, when we say “intelligent control,” we simply mean a control system that does more than execute a numeric control loop. It may, for example, interface to complex subsystems and incorporate complex reactions to external events. These systems require both significant software effort and control systems mathematics. ControlShell targets these complex systems that require significant software in addition to mathematics.

ControlShell is useful for any complex control system. It is especially attractive for electromechanical systems such as those found in aerospace, military, industrial automation and robotics applications. Many successful applications fall into the category we affectionately label “BWS,” which stands for “Big Weird Stuff”. These large, one-of-a-kind systems, dominated by very expensive custom coding efforts, benefit greatly from ControlShell’s development capabilities.

Software Modeling Tools. Software modeling tools built for other primary challenges—such as telecommunications or databases—are not well suited to the challenges of complex intelligent control systems. Generic programming languages and methodologies like UML, ROOM, or SDL are nice for analysis and modeling. They carefully define the nuances of syntax and meta language. But they often don’t truly accelerate development.

Dynamic System Design Tools. Dynamic system control and simulation tools like Matlab/Simulink or MatrixX are also useful. They allow you to build and test dynamic models of the physical processes in your system. However, in most systems control is only a small percentage of the total problem. Simulation tools like Simulink or SystemBuild are not good programming systems, and thus can’t handle truly complex systems that require significant custom coding.
While ControlShell includes powerful mathematical control capability, it is not usually the best choice for problems that are dominated solely by difficult mathematical control. For problems that combine both complex software and difficult control design, ControlShell is integrated with The MathWorks’ Simulink and Real-Time Workshop products.

**The Solution.** To quickly build control systems, an architecture must support both low-level sampled-data systems and high-level strategic control. It must encourage reuse, because reuse is the key to leveraging software across teams, organizations, and industries. It must be sufficiently flexible to encompass the wide array of issues facing complex real-time systems. It must allow for easy testing and maintenance. It must support real-time distributed computing.

These are difficult problems. Many are unique to complex electromechanical or intelligent control systems. ControlShell is specifically designed to solve them. It provides a tool set that changes the way complex software systems are written. Instead of detailed analysis followed by laborious inheritance diagrams followed by hand coding, ControlShell provides a ready-made, proven architecture that fits many problems in the specific domain it targets. ControlShell does not try to address the needs of every application. But where it does apply, ControlShell radically improves the development process.
2 ControlShell is an Application Framework

So what is ControlShell? ControlShell is an application framework. An application framework is a development environment, encompassing all the tools and glue required to build applications. It includes an object hierarchy, a set of tools, and pre-written software components. Teams use the framework to coordinate their efforts, build and test software and hardware, and deliver the working application.

Application frameworks reduce risk and accelerate development. Every complex application needs a framework. You usually have to develop your own, despite the expense, risk and maintenance nightmare. In fact, you probably have one in house now that is—despite a lot of work—aging rapidly. A framework that fits your problem is a huge win. With the right application framework, you are already halfway done with your development.

ControlShell is the first commercial-off-the-shelf (COTS) framework designed for control. ControlShell’s object hierarchy is the result of extensive analysis of the structure needed to build complex control systems. It has been proven in many applications, from lightweight embedded vehicles to large distributed control systems. It includes a targeted object hierarchy, graphical productivity tools, run-time libraries, debugging and analysis tools, and a library of controls-relevant software components. ControlShell is the right solution for complex control and simulation systems.

This section examines the processes required to develop complex control and simulation systems. We show that there are two ways to view a complex system: top down and bottom up. We argue that these views must be combined, but they cannot be combined without a specific application framework that fits the problem domain. We then overview the ControlShell framework.

2.1 An Analysis of System Design

2.1.1 Top-Down Design

“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; it can eventually solve most problems. It is also intuitive, it allows division of labor, and it encourages rapid prototyping through consideration of 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.

Most people naturally think of complex systems in a top-down, object-oriented fashion. For instance, when you describe a car, you think of the engine, the frame, the transmission, and other systems as functional, complete objects that cooperate to make the whole complex system.

However, the top-down design process has a serious flaw: it leads to unique solutions to every problem. Each top-down design subdivides the problem in a unique way. None of the resulting subsystems can be reused easily. For this reason, top-down design has been called “the process of making all your worst mistakes first.”

A top-down designer building a car would start with a concept drawing, and break it into subsystems as best fit the problem. However, like a concept car, the result will contain many strange parts that all have to be painstakingly handcrafted.

2.1.2 Bottom-Up Design

Bottom-up design is the process of synthesizing from pre-existing components. A bottom-up design process for building a car would be to walk into an auto-parts store and begin piecing together the vehicle from parts on the rack. Bottom-up design is great for reusing components. However, without the overall

---

1. Professor David Cheriton of Stanford’s Computer Science Department.
picture and goal in mind, it often does not result in a clean division of functionality. The bottom-up design process provides no overall view to communicate the goals and progress of a project between team members.

However, bottom-up design is critical for reuse. An analogy with component-based hardware design is instructive. Chip-level board design is a well-established technology. The components (chips) each perform a well-established defined functionality. Most are quite simple. Complex parts evolve through much experience to be truly reusable. Nobody tries to design a board top down; the result would be requirements for many types of chips that don't exist.

Of course, software design is not identical to hardware design. Software is much easier to create and modify. However, to achieve true leverage, software must be reused unmodified. Copying and modifying code creates many source versions that must be independently tested, maintained, and supported. Only true reuse generates the leverage that results in large dividends.

To be truly powerful, bottom-up design must support multiple levels of granularity. Small, functional components are easy to reuse. However, they don't provide much benefit. For example, a component that returns a simple mathematical function such as “add()” is quite general. However, it takes many components of this scale to build a complex system. On the other hand, large-scale components provide large-scale benefits. The reuse of an entire subsystem as a component would greatly accelerate the next system's implementation. But complex components have complex interfaces and specific functions. The more complex the component, the harder it is to reuse. Clearly, to maximize the benefits of component-based design, you need to reuse components at many levels. You also need a clear structure and clearly-defined interfaces.

Much reuse is also lost due to mechanics. Components must be easy to find, incorporate into new designs, test and maintain. Specific tools for building, browsing, and managing repositories of components must be provided.

Still, components, and the reuse they foster, are immensely powerful. Components speed your development by leveraging already-written code. Components are better-tested, higher-quality software. They allow software to easily span teams, projects, companies, and even markets. The promise of component-based programming is huge.

2.1.3 Top-Down / Bottom-Up

In recognition of the trade-off between top-down and bottom-up design, ControlShell has taken a unique merged approach. ControlShell allows you to start with a global, undefined concept, and decompose it into more specialized subsystems. Along the way, you define and incorporate reusable interface objects that modularize the design and insure clean divisions between subsystems. Finally, you build your subsystems from the “bottom up” by graphically combining components from a repository of reusable software. This merged approach combines the power of object modeling with the leverage of component-based synthesis.

The top-down / bottom-up combined approach is very intuitive. It allows designers to work simultaneously at different levels of abstraction—today on the overall design, tomorrow on a specific subsystem. That makes it easy to iterate between partially-specified designs and fully-specified subsystems. Rapid iteration at different levels of the design is a key enabler of powerful, clean systems.

However, it isn't trivial to do. The two approaches are very different. They require fundamentally different thought processes. Merging them requires structure. It requires a specific, well-defined object model and keen attention to module interfaces. ControlShell's tool set provides this structure.
2.2 ControlShell’s Framework

2.2.1 Interfaces are Key

Interfaces are the key to modular design. They are also the key to merging top-down design with bottom-up synthesis. During the top-down decomposition, interfaces must be defined between modules. As long as the interfaces are unique, top-down decomposition must continue. When the design is refined to the point that existing interfaces can be used, pre-built components may be used to fill in the module functionality. Thus, the goal is to divide the problem from the top down, but strive to reach commonly-defined interfaces along the way.

In the past, there was no mechanism for this to occur. ControlShell is the first tool in this domain that provides careful and explicit definition of interfaces. Perhaps more importantly, ControlShell interfaces are reusable objects. Thus, it provides a mechanism for both defining and reusing interfaces. Because this mechanism exists, standardized interfaces are possible. Over time within a project, an organization, or even a market, these interfaces allow component-based design.

Since these interfaces can be attached to many different software objects, they provide a layer of abstraction between the external behavior of an object and its implementation. This provides a new paradigm for programming, and enables massive software component reuse.

2.2.2 Event and Cyclic Processing

Most control systems have two very different aspects: cyclic data processing and event-driven reaction. In fact, these two traits are common to most complex systems. They are the most fundamental functionality in real-time electromechanical systems.

For instance, your body processes sensors from its surroundings and maintains itself. Each instant, you adjust your balance, you maintain your temperature, and you control the position of your arms and hands. These functions are performed continuously, based on your sensor inputs—what you see and feel. You also process events and execute strategies. If someone throws a ball at you, you will immediately evaluate what it means and formulate a strategy. Should you duck? Reach out and catch it? Ignore it? If you decide to catch it, you have to compute an intercept trajectory, reach for the ball, sense when the ball contacts your hand, and close your hand. This sequence of steps is event driven; each step begins when some event is sensed. So, a continuous process controls the actual motion of your arm, but the strategic plan is event driven.

A robot arm behaves in the same manner. At regular sample intervals, the controller reads the position and velocity sensors, computes where the arm should be, and outputs commands to the motors to drive the arm on its way. 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. Because it concerns data flowing through the system, ControlShell calls this dataflow.

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. Each step in the sequence is begun and terminated based on the arrival of an event in the system. Because this is done naturally with a finite state machine architecture, ControlShell calls this state programming.

2.2.3 Composite Object Groups (COGs)

In complex systems, event and continuous processing are tightly integrated. On closer examination, we find that this structure pervades and defines the system. We claim that most real-time systems can be divided into recognizable units—subsystems—that include both event and continuous processing. For instance, a robotic system may consist of subsystems for the arm, end-effector (e.g. a gripper), and a vision system. Each processes data continuously, but also executes sequences when events are detected. This observation is critical; we’ll examine it further below.
ControlShell codifies this structure. ControlShell systems are built from objects that contain both event-driven and cyclic subsystems called Composite Object Groups, or COGs. COGs provide explicit, graphical integration of both types of processing. They are fully hierarchical—you can design complex systems by building COGs made up of other COGs. Figure 2 demonstrates a typical structure.

A COG may also contain a third type of component, an interface. Interfaces specify both the data flow through the COG and the set of methods the COG uses. Interfaces may be attached to any object within the COG. They may be internally connected, or exported to be connected by another COG. There are two types of elements in interfaces: “pins” that specify data connections, and “bubbles” that indicate methods. The interface can call or implement a method; the bubble icon indicates which direction is active. Figure 3 contains an example.

**2.2.4 Combining Procedural (Functional) and OO Design**

COGs combine event-driven and cyclic processing. COGs also perform another important function; they bridge procedural and object-oriented design.
Both event-driven and cyclic programs are most naturally thought of procedurally. For instance, a typical data flow function might be a PD controller. The controller inputs sensed and desired values, and outputs a command to be sent to an actuator (such as a motor). It thus performs a well-defined function on external data. Controls engineers are used to thinking in this manner; control systems are built from blocks that process inputs and produce outputs.

On the other hand, larger subsystems are most often best modeled as objects. Objects contain internal data and functions. They have more complicated interfaces to the external world. They are usually thought of as executing independently of and concurrently with other objects. They communicate by exchanging asynchronous messages. Object-oriented design is fundamentally different than procedural thinking.

A COG is an object that is built from procedural components. It neatly bridges the gap from the procedural thinking that is efficient at the low engineering levels to powerful OO thinking at higher systems design levels.

COGs are remarkably powerful. They can model simple subsystems, yet combine into large hierarchies that control extensive, distributed systems. Even in large systems, COGs remain intuitive structure. COGs are unique to ControlShell; RTI has a patent pending on the architecture.

### 2.3 Overall Architecture

ControlShell’s overall architecture is diagrammed in Figure 4. On the left, a dataflow system is generated to control the arm of a robot. Similarly, a state machine is built to control the activities of the arm. These functional subsystems are combined with an interface into the Arm COG in the center of the figure. This COG is combined with others to build the robotic system, which in turn is part of a workcell.

The left side of the diagram is entirely functional 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.
2.4 A Note on Object Modeling

Objects model systems by implementing the behavior required of each part of the system. General modeling languages such as C++ and UML provide a means of describing arbitrary relationships between the various system objects. Each object can contain arbitrary functionality. ControlShell makes no attempt to be a general system-modeling tool.

Instead, ControlShell makes a bold but powerful claim: most control and electromechanical systems can be modeled as systems of COGs. The utilization of a specific object model bestows many benefits. For instance, it shortcuts much of the tedious analysis and object-hierarchy design phase. It allows us to provide specific structures and reusable objects that fit specific problem domains. It really gets the “pedal to the metal” and gets complex code working quickly.

Comparison to UML. The Unified Modeling Language (UML) is an object-oriented graphical notation that arose from the combination Booch, Raumbaugh, and OOSE methodologies.

ControlShell is not like UML at all. UML is a very general modeling language. It is used for everything from airline reservation transaction systems to cellular phone firmware. 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. ControlShell is based on the premise that it is possible to build many systems from components that fit into a common structure.
It is precisely the existence of common structure that lets you reuse software, utilize tools, and have powerful run-time services. This is the essence of building systems from components. Programming paradigms have always balanced this power-versus-generality trade-off. Each application should utilize the least-general—and therefore most powerful—paradigm that solves the problem (See Figure 5).

ControlShell is not as general as the UML tools. It cannot solve as many diverse problems. You will never see ControlShell being proposed to configure Oracle databases or build a telephone switch. ControlShell is focused on complex real-time control systems. Period.

By focusing on a particular problem domain, we gain immense power. We can provide a framework that defines your problem so you can begin immediate implementation. We provide run-time facilities that solve large parts of the problem. We can automate most mundane things, while retaining the flexibility to accommodate complex system development. We can provide tools to develop reusable interfaces, and use those interfaces to build reusable modules. We can provide pre-built component libraries of many types of objects, in a form sufficiently structured to allow immediate use in your application. We can also give you this power; you can build your own libraries of reusable components.

In summary, UML is a language that allows you to analyze your problem, build a structure to solve the problem, and then implement that solution. ControlShell is an application framework that lets you directly implement a solution. You can start at a high level and develop an overall system plan. However, you build your plan with well-defined structures that we (perhaps arrogantly) claim are sufficiently general to solve many types of complex control problems. You use our structures (dataflow diagrams, state machines, COGs, interface objects) to build your system. If ControlShell fits your problem, you are light years ahead.

---

**Figure 5  Generality-Power Trade-off**

*There is a trade-off between power and generality in any implementation technique. Assembly language is very general, but inappropriate for many problems because it has little structure. The optimal implementation uses the most powerful structure available that fits the problem.*
3 ControlShell is the Right Solution for Intelligent Control

In this section, we examine what makes intelligent control systems hard to develop. We review the specific facilities needed to address these problems. We then show how a focused tool and application framework can greatly accelerate development while reducing the risk of “do it yourself” designs.

3.1 Proven Real-Time Framework

It takes time to build a good real-time software structure. The required expertise is rare. Without a lot of experience, or even with it, the design is risky. Many designs “run out of steam” only after months or even years of development.

Building real-time software for control systems is even trickier. Control systems have needs that differ from other systems: feedback control, sequencing, device interfaces, and more. There are excellent control and simulation tools, but they don't address complex programming needs and thus solve only a small part of the problem. There are general programming tools, some even designed for real time, but they don't provide any really useful facilities for control problems. What's missing is a true programming system with pre-built architectures designed specifically for the right problems.

Obviously, if you can use a proven structure, your risk is decreased. You can spend more time solving your other problems. Your project will have more consistency. Moreover, a structure provides other benefits. It allows libraries of generic components to be developed. 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.

3.2 Fit to Complexity

Systems can be complex in many ways. Some systems are complex because of the mathematical algorithms they contain. For instance, despite its single input, a speech processing system is a good example of mathematical complexity. Other systems are complex because they contain many concurrent elements. For instance, a telephone switch must make many connections. Each connection is simple, but it must make thousands of them per second.

Intelligent control systems exhibit both of these types of complexity to some extent, but more importantly are faced with computational structure complexity. These systems must make strategic decisions, interface to complex intelligent modules and user interfaces, process data, execute feedback loops, change modes, and drive complex hardware. Interconnecting subsystems leads to even more structural complexity. Many systems must also be modal, changing behavior in response to changes in operating conditions. Modal behavior can add very significant—and often unforeseen—structural complexity. Structural complexity dominates most intelligent control systems.

The software architecture must utilize structure to manage this complexity. Since it is difficult to handle all types of complexity, architectures must make choices based on their target systems. There are tools on the market to address mathematical and concurrency complexity. ControlShell provides specific facilities to address structural complexity. It does not focus on concurrency management, because to do so injects artificial concurrency where none is natural. It recognizes the duality of cyclic and event processing, and does not try to fit event processing into a fundamentally cyclic structure or vice versa. The resulting design is simple, powerful, and maps directly to the electromechanical system problem.

ControlShell includes specific tools to support interface capture and reuse. It provides sampled-data system and run-time engine to enable advanced feedback control. It permits graphical strategic sequencing with a powerful state-machine programming system. It also provides for distributed systems, dynamic simulation, and modal control, all functionality complex systems often need.
Complex systems often require custom coding. To support this, ControlShell is fundamentally designed as a programming system. It doesn’t try to eliminate all your coding. Instead, it provides a framework in which you can build your system. ControlShell strives to provide the flexibility that inventive engineers and programmers need to solve tough and unique problems. Its extensible open architecture won’t cripple good programmers.

3.3 ControlShell’s Structure
We will now examine some of ControlShell’s specific structures.

3.3.1 Atomic Components
All ControlShell systems, at the lowest level, are built from Atomic Components. Atomic components are directly implemented in C++ code. Atomic components are created by first specifying their interfaces with the Atomic Component Editor, shown in Figure 6. This editor specifies the data interchange and method signature for the component, as well as internal methods that this component will implement. It then generates a code template for the component.

![Figure 6 The Atomic Component Editor](image)

The ACE is a front end to the code generators. It allows you to specify the interfaces, methods, and all other aspects of both user-written and RTI-provided source code modules.

3.3.2 Sampled-Data Systems
We have already seen one of the most important structures that ControlShell supports, the COG. Event-driven sequencing and sampled-data processing are the two most pervasive computing requirements of real-time control systems.

A sampled-data system is one that regularly gets input data, processes it, and then outputs the results. This class of system includes most feedback control and signal processing systems. Sampled data systems are extremely common in electromechanical designs.

Because it deals with data flowing between blocks, ControlShell’s sampled-data structure is called the dataflow system. COGs may contain dataflow systems, both alone and integrated with state machines.

Dataflow Components. ControlShell builds dataflow systems from small, reusable objects called dataflow components. Dataflow components are a derived type of atomic component; they are directly implemented in C++ source code. Dataflow components read input signals, generate output signals, and use
reference signals. Input and output signals pass data to and from the component. Reference signals are often used for parameters such as gains or file names. Figure 7 shows an example of the pdControl component that implements a simple Proportional-Derivative controller.

![Image of an example component diagram]

**Figure 7 Example Component**

This is a COG with 5 input pins and two output pins. It also exports (implements) three externally-callable methods, bundled into an interface.

**Dataflow Systems.** Dataflow components are linked into systems with a graphical editor known as the Composite Object Editor (COE). An example diagram, a portion of a controller for a multi-link robot, is shown in Figure 8. Each block in this diagram represents either a dataflow component (a C++ object), or another, lower-level COG.

A dataflow diagram is executed once when its clock period expires. The blocks are sorted and executed in dependency priority order. Thus, blocks that produce outputs used as inputs to other blocks are run first. This allows signals to propagate between blocks with minimum latency; each clock period the signals produced by the first blocks will propagate all the way through the diagram.

The frequency of the clock period is often called the rate. ControlShell dataflow diagrams actually support multiple rates attached to arbitrary periodic events as clocks. However, that functionality is beyond this discussion.

**Real-Time Matrix Library.** ControlShell’s dataflow system is based on a real-time matrix library called the CSMat library. CSMats are named arrays of values. They are usually floating point numbers, but can actually manipulate many types of data. All the common matrix operations, including arithmetic, min/max, various splines, and more are supported. On-line matrix inversion and multiplication are also supported.

The CSMat library is unique in that it is expressly designed for real-time systems. Written in C++, it fully supports natural language syntax, such as C = A+B. However, it is extremely careful with non-deterministic actions. For instance, the library itself never allocates temporary storage, because memory allocation is not a real-time operation. Any required storage is allocated during initialization. The operations carried out by the library are always deterministic and time bounded.

**Types.** Structured types are an important way to encapsulate data and pass information between modules. However, types pose a significant threat to usability. The only way to guarantee that components can be mixed and matched without type conflicts is to use only a single type.
The mathematical modeling tools (such as Simulink and SystemBuild) do exactly this; the only type is a floating-point value, or possibly an array of values. That is not an acceptable option for a programming system. In ControlShell, most data lines are CSMats. However, data lines may represent any structured type. When a new type is defined, a code generator automatically creates a component that splits the structure into its elements, and another component that merges elements into a new structured type. These merge and split components can be used to interface any components that use the type to components that do not understand the structure. ControlShell thus provides type encapsulation without sacrificing reusability.

**Dataflow Component Repositories.** One of the great benefits of using a structured framework is leveraging from pre-built components. ControlShell ships with many dataflow components and COGs. Examples include controllers (e.g. PD, PID, and LQR), estimators (LQ, Kalman filter, dead reckoning), motion generators (smooth splines, “bang-bang” motions, and geometric path-following), filters, integrators, matrix utilities, device drivers, kinematics, dynamics, and many more. All components are shipped as source.

**Simulation.** ControlShell also supports dynamic simulation of all systems. Except for the fact that it does not run on a real time operating system, the simulation environment matches the run time environment, and therefore has all the tools and facilities available. The simulation system is also set up to be very flexible. You can easily “mix and match” actual hardware execution with simulated systems on a COG-by-COG basis. Thus, for example, it's easy to test parts of a complex system, while simulating the rest. (See Section 5.3).

### 3.3.3 Event-Driven Systems

All complex systems must be strategically guided. Since real-time systems must operate in a complex, event-rich environment, strategic control is the process of reacting to events. Traditional, sequential programming is not well suited to managing events. Event-driven programming—the process of writing actions that are executed when specific events occur—is much more appropriate.

---

Figure 8 A Robot Arm Controller

*This is a simple feedback controller for a robot arm. The sensor input, measured positions and velocities of the arm’s joints, come in at the upper left. The large kinematics module calculates the end-effector position from this data, and a matrix known as the Jacobian. This matrix is used to compute the output to the motors. The diagram shows that problem-specific blocks are easily combined with standard components, such as the pdController above.*
The most intuitive way to build event-driven programs is with a state diagram. A state diagram shows blocks (states) essentially representing where the system spends time waiting for an event. Possible events that could occur while waiting and the actions to take in case of those events are attached to the states as a graph. The diagram thus represents an automaton known as a finite state machine (FSM).

ControlShell’s FSM model is essentially similar to the UML state machine model. ControlShell’s state machine model extends UML to more directly target control systems. For instance, ControlShell provides features such as safety overrides and explicit data input and output. In addition to UML-style event-driven transitions, ControlShell also supports integrated numeric transition expressions. The numeric expressions are periodically-evaluated, thus permitting synchronous clocked-state-machine behavior similar to dynamic-system-design tools such as Simulink. ControlShell’s FSM is also component-based; it makes both transition routines and state diagrams more easily reused. The details of the state machine model are beyond the scope of this paper, see the manual for details.

The simple state machine in Figure 9 shows some of the power available with this model. This diagram is a reusable state subroutine that will cycle an output (called desiredPos) between a HighValue and a LowValue. The steps occur when the measured position (pos) is near enough to the current target, and measured velocity (vel) is low. Note that this is a very useful function: it allows any control system to be repeatedly stepped as a test.

The diagram in the figure also demonstrates mixed event-driven (asynchronous) and clocked (synchronous) behavior. For instance, the “stop” transition will turn off the stepping whenever it is received. On the other hand, the steps occur when the clock-driven evaluation of the position and velocity are within the range required for a step. The figure itself is a reusable subroutine. It utilizes a reusable state transition component (called setValue) that sets the output value.

Figure 9 State Machine

*State machine programming allows simple strategic programming. ControlShell’s FSM model is powerful, it extends Harel statecharts by adding stimuli with memory, reusable subprograms, reusable action routines, and optional synchronous clocked behavior.*

The diagram in the figure also demonstrates mixed event-driven (asynchronous) and clocked (synchronous) behavior. For instance, the “stop” transition will turn off the stepping whenever it is received. On the other hand, the steps occur when the clock-driven evaluation of the position and velocity are within the range required for a step. The figure itself is a reusable subroutine. It utilizes a reusable state transition component (called setValue) that sets the output value.
3.3.4 Mode Control

Complex systems are usually modal; i.e., as they execute, they need to change their behavior in response to changes in their environments. Changing modes can be a very complex operation. In fact, many real-time system architectures “run out of steam” and must be replaced at great cost because they failed to account for the vast global implications of changing modes. The problem arises because each subsystem may have to adjust to the change. In other words, each subsystem also must be modal. Worse still, these subsystems must adjust to each other’s changes. This creates a strongly-interconnected system; a mess that cannot be handled by these architectures.

For example, consider a satellite system with several subsystems. Suppose the sun sensor has two modes: seeking and tracking. The attitude control system may also need modes, depending on whether or not it has a sun sensor to rely upon. Further, the power management system may depend on the controller’s modes. If you extend this to all the subsystems involved, and consider the effect of cross dependencies, you can see that there could easily be hundreds of total system operating modes. This is not unusual in complex systems.

There are many ways to account for changing modes. Some systems pepper the code itself with “if” statements. This works in some simple cases, but soon results in “spaghetti” code. Moreover, the modal behavior is hidden from the developer and the team.

Some systems provide “switches” that enable or disable subsystems. While this does make the modal behavior visible, switch blocks and their connections rapidly dominate complex systems. Small mode changes—those that affect only a few blocks in a large subsystem—cannot be supported easily. Systems that must support many similar modes become hopelessly complicated. Worse, it’s almost impossible to look at a diagram full of switches and tell what’s really going on!

ControlShell takes an entirely different approach. Each subsystem can support any number of named modes. Defining a mode for a system simply requires defining a mode for each of its subsystems and assigning it a name. Modes are defined graphically and mapped down the hierarchy via a unique patent-pending technique. Thus, the active blocks in a mode are easily displayed. Small changes in a configuration only require changing the few affected subsystems and assigning it a new name. These named modes will allow the system to switch operational modes very rapidly at run time.

You can think of swapping modes as throwing switches in the dataflow diagram. This implementation, however, is much more powerful than switches, since swapping mode configurations can enable and disable entire constellations of components, and each function can have many implementation “switch positions”. It would take an unmanageable number of switches to perform things that a simple configuration swap can achieve. Moreover, the configuration gives a graphical display of exactly which components are executing in which modes.

Modes are very powerful when combined with ControlShell’s finite-state machine engine. In response to events in the system, the state machine can rapidly change the mode of operation by using named modes.
4 ControlShell Fits Naturally Into Your Team’s Process

Too much time is spent in building systems. Long meetings explaining what you’re doing and what progress you’ve made, tedious work on repetitive tasks, and excessive effort in documentation wear everyone down. Much of this can be alleviated by a toolset that truly fits the team and the team’s process.

4.1 The Team Programming Challenge

To be effective, development teams face four primary challenges: coordination, communication, modularization, and reuse. This section examines each of these issues, with special emphasis on challenges facing real-time systems.

4.1.1 Coordination

Coordinating 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.

Tools for Each. ControlShell recognizes that development teams contain a mix of skills and talents. Each team member must be empowered to understand the problem and contribute his or her expertise efficiently. Therefore, ControlShell provides tools for talented programmers as well as for the system and domain engineers who do not want to write code.

ControlShell’s visual tools leverage the skills of everyone—system architects, programmers, domain engineers, and end users. They provide a common language that ties the team (and even management) together. The graphical tools also provide reliable, easy-to-understand documentation that is guaranteed to be up to date.

ControlShell’s interface design system lets system-level architects define interfaces and modules. Once defined, the interfaces are reusable objects that can be used throughout the system. With the code generators, device and algorithmic programmers can build software components that require custom coding. The system domain engineers can build complex systems from these software components with the graphical diagram editors. Finally, test and integration specialists can use the array of run-time tools to build and test working systems.

4.1.2 Communication

As soon as a project grows beyond the capability of a single individual, communication becomes a critical problem. All team members must both understand and contribute to the project. Graphical diagrams communicate overall design much more effectively than any textual or spoken description. They must be commonly understood by all the team members, intuitively fit the problem, and be sufficiently expressive to describe the problem.

These are stringent requirements, especially for a 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. The graphical notation must fit the specific problems faced by the team.

Intuitive Diagrams. ControlShell’s diagrams combine the procedural “input/output” diagram familiar to engineers with an object-oriented viewpoint powerful enough for complex software. The engineering design diagrams directly merge with an open software programming system. The graphical diagrams are straightforward, designed to provide a common language that all team members can use and understand.
The hierarchical design helps new users (and managers) understand the system without being overwhelmed by details. Specific tools are provided for interface design, system configuration, and documentation. For the target applications—complex engineering systems that require significant custom code—ControlShell provides an expressive language that all can understand.

4.1.3 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 non-trivial. 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.

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. ControlShell’s well-defined structure of hierarchical COGs with clearly-designed interfaces provides this structure. A software architecture and toolset that match your problem is an amazingly powerful resource.

4.1.4 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.

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

Figure 10 Repository

Repository-centric design pervades ControlShell. Everything is stored in a repository. RTI provided components, user-generated components, and even “top-level” applications are stored in a repository for potential reuse.
Reusability. ControlShell has a penchant for reusability. Virtually everything is placed in reusable libraries known as repositories. Team members use the contents of the repositories to share their work. Developers write new components and store them in repositories. Engineers pull together systems built from components in the repositories. Even the final subsystems are stored in repositories for use in prototyping, testing, implementation and maintenance. The repository-centric view of the world provides a reuse mindset that pervades the entire project.

You may have noticed the file-system-like tree view on the left side of all the diagrams in this document. In case you missed it, an expanded view is provided in Figure 10 The tree view is a repository browser. You configure your development environment to load any repositories you need: from RTI, our other customers, or your own organization. Then you can utilize any item—from primitive component to full subsystem—as needed. The repository really does contain most everything, including all the atomic components, dataflow components, state transition action routines, dataflow diagrams and systems, finite state machines, interfaces, COGs, and more. In fact, you are never developing a “top level” system or throwaway code module in ControlShell. Everything is potentially reusable.

4.2 Your Team’s Process

Your team, big or small, has a current process. ControlShell fits naturally into that process and enhances it. It manages each of the steps, from system analysis and prototyping through test and integration to maintenance and upgrades. By addressing the need for coordination, communication, modularization, and reuse, ControlShell makes your process run smoothly.

For instance, a typical process for an aerospace system development probably goes something like this:

- Someone (usually with a controls and simulation background) prototypes the system for proof-of-concept during the proposal stage
- Once you win the project, the controls team designs a control system
- Systems engineering passes the design over to the software group for coding and manages the resulting software to a paper specification
- The software is sent over to the simulation group for testing and integration with hardware
- Finally, someone tests the shipping system.

All of this is done with few tools. Handoffs between groups are significant friction points. Overall, the process is slow, prone to errors, and painful.

Let’s break this down further, and look at how ControlShell helps.

Prototyping and requirements capture. ControlShell is a great prototyping environment. Many systems can be prototyped quickly by reusing previous code modules or selecting from RTI’s libraries. The fast development cycle makes incremental changes painless. ControlShell is also integrated with DOORS, the industry-standard requirements capture tool.

Systems analysis and design. Most large projects start with systems analysis. Keeping in mind the highest-level requirements, system engineering breaks the problem down into manageable pieces. With ControlShell, you can specify a single system block and then graphically break it down into subsystems. In doing so, you develop well-specified interfaces between the subsystems. You then use the interface objects developed to communicate the design to the other teams. The handoff is graphical and seamless.

Control design. If your system includes a sophisticated controller, you will likely use a dynamic system design tool like Simulink/Matlab to design the control algorithms. When that process is complete, you can generate the algorithm code and automatically drop it into a ControlShell component. Of course, you can develop simpler control algorithms directly in ControlShell. ControlShell can simulate your system on the host as well.
Coding. Once the system design is available, the teams need to implement the software for the subsystems. ControlShell can be used to develop most all the code. For instance, you can develop ControlShell components to communicate over a network or some other communications channel (like telemetry and command on a satellite). You can write device drivers to interface to the hardware. You can develop payload interfaces and software. You can also easily include legacy code and interface to pre-existing subsystems. Clear, usable interfaces are a great boon here. The interfaces are automatically managed in the same way as any component. Interface specifications are automatically checked against both the system design and the developing subsystems.

Simulation and test. ControlShell lets you do real-time, hardware-in-the-loop simulation with “mix and match” software and hardware subsystems; you can easily define different configurations of hardware/software subsystems to test (See Section 5.3). For instance, you might simulate with a software model of a sensor today, and then swap a device driver and test against the actual hardware tomorrow.

Integration, delivery, maintenance. Finally, you need to deploy the software on the deliverable hardware and do final test and integration. You can do that with your development system, and then finally generate all the code for the final system and embed it.

When you deliver the system, ControlShell’s tools help with install and debug. All the debug tools, including the command-line shell, data monitoring, state-machine tracking are available on the running system so you can iron out the last minute site integration bugs. The tools can even be used remotely, so you can help debug from the home office.

Finally, you need to maintain and upgrade the system. To change a piece of the code, just swap out a component. You can even do it without stopping the system. The debug tools can be used whenever they’re needed.
5 ControlShell Speeds Your Development

Even given a great framework, you can’t get work done quickly without the structure and discipline of great tools. Without visual tools, the design is poorly understood. Without automated development, precious time is spent writing glue code and doing menial work. Without good debugging and visualization, you spend countless hours trying to understand what’s going on. In this section, we will explore the specific features and tools ControlShell includes to accelerate and enhance your development process.

5.1 Code Generation

Perhaps the most important objective of an architecture is to promote understandability. A system that is easily understood is quicker to learn, faster to develop, easier to debug, and less costly to maintain. It is inherently flexible because users can understand how it works and modify it for their needs.

ControlShell focuses on understandability. Its fundamental paradigm is “What you see is what you execute (WYSIWYE):” every diagrammatic component maps directly to an executing object. The overall build process is detailed in Figure 11.

ControlShell implements WYSIWYE with the component-level code generator and dynamic linking technology. A C++ class definition is generated for each atomic component independently. These are compiled as separate, small, independent modules.

The graphical diagrams are never translated into source code. Instead, the run-time ControlShell engine reads the same description files that the graphical editors use to store the diagrams. From the information in the files, individual compiled code modules are linked dynamically at run time into a working system that matches the graphical diagram. This design provides true WYSIWYE functionality—every module on the diagram corresponds to an actual C++ class definition and an instantiated object at run time.

Tools with graphical compilers, by contrast, generate monolithic, incomprehensible code. A block in the diagram may be reduced to lines 738 through 843 of a single 3000-line subroutine. This code is difficult to understand and debug, even on a good day.

With component-level code generation, users have full control over the internal implementation of components. Since each module is generated independently as a separate class, the source is readily understandable. Advanced programmers can build their own classes from the automatically-generated base. RTI ships full source for all the components in the standard library to be used as examples or as the basis for new designs.

The WYSIWYE design has other benefits as well:

- **Fast Cycle Time.** Since the diagrams are not compiled, you don't have to recompile to test changes. That greatly accelerates the development cycle.

- **Interconnectivity.** All the objects in the code are stored in an on-line hierarchical database. Any code in the system can look up other objects at run time. Objects can implement menu routines, external methods, and many other facilities. The design allows levels of integration that are only otherwise available by writing all the code yourself. ControlShell’s run-time tools take advantage of this. You can poke & prod at components, change parameters, start and stop module execution, stimulate and modify the state machines, and watch most any variable change, all at run time.

- **It's easy to wrap legacy code.** The module-level code generator allows linking with existing software and modules without violating the architecture. External and legacy code can be included by simply calling the application interface from within any ControlShell component. Since a ControlShell class wraps the code, the impact on the rest of the system is well controlled and understood.

- **Minimal Global Impact of Local Changes.** Finally, systems with graphical compilers must regen-
5.2 Fast Development Cycle

ControlShell has a blazingly fast development cycle. ControlShell’s component-level code generation, mixed-mode simulation, run-time support and quick iterative design cycle increase team and individual productivity. Installation, code generation, makefiles, system building and documentation are all auto-

---

Traditional Graphical Programming Process

ControlShell’s Programming Process

Figure 11 Build Process

ControlShell’s build process differs radically from traditional graphical programming systems. Traditional systems, shown on the top, compile diagrams built from vendor-supplied blocks into monolithic source files. This source is then compiled and loaded onto the target. A change in the diagram requires a new compilation cycle and is therefore painful. Adding user-generated blocks is often awkward.

ControlShell, shown on the lower half of the figure, lets you build diagrams from user-generated, vendor-supplied and third-party components. Each component is individually compiled and linked by the target system when the run-time system initializes. The compiler need not be used when the graphical diagrams change. The task cycle—from editing the diagram to running the system—requires no compilation and is thus very fast. When compilation is needed, only a few small objects must be built into objects. Moreover, the generated code consists of small, well-defined objects, and is thus easier to understand.

In the case of traditional systems, generating code for the entire system when only a small change is made to the diagram. For safety-critical systems, that requires re-evaluation of the entire code set. With ControlShell, the only code that changes is the code you edit. The diagram linking impact is limited to just the portions of the diagram that actually changed.
Late binding means you can test diagram changes without recompilation. On-line debugging and analysis let you see and test the system while it runs. ControlShell speeds the transitions between design, execution and debugging.

5.3 Mix and Match Simulation

ControlShell provides the capability to define “executable systems” by assigning specific components to each system. During final translation from ControlShell design to executable code, the designer specifies which “executable systems” to create; each system results in a separate executable image.

This mix-and-match capability is ideal for HIL test applications. For instance, the designer can define “hardware” and “simulation” implementations for each major subsystem. The “hardware” implementation may include device drivers to exercise the actual system. The “simulation” implementation is a software simulation of the hardware’s function. A ControlShell executable is then built by simply selecting the “hardware” or “simulation” option for each subsystem. In fact, the designer can generate as many executables as needed to match each possible combination of hardware/software subsystems.

### Table 1 Mix and Match HIL Simulation

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine Control</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GPS</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Inertial Navigation</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Pilot Interface</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Wing actuators</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Rudder</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Flaps</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Autopilot</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FMS</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Instrument Interface</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

This table shows a possible configuration for an HIL test system. Many other configurations are possible. The graphical tools make it easy to mix and match real and simulated subsystems.

The definition of systems is done from the ControlShell graphical user interface, so they are very easy to define. Note in particular that selecting between alternative “Executable Systems” for a single subsystem does not impact any other subsystem. All other subsystems are guaranteed to be running the same code.

5.4 Superb Run-Time Support and Debugging

ControlShell is designed as a hands-on system. It includes an on-line menu and command shell. Most run-time engines and many components implement menus that allow users to control the execution as the system runs. Facilities are provided, for example, to:

- View and modify the object database,
- View and change the value of any variable,
- Enable components, COGs, or configurations,
- Change sample rates,
- Start and stop the system or any subsystem,
o Install data to be monitored by StethoScope,
o Reorder module execution sequences,
o Send test stimuli to state machines,
o Turn debugging verbosity levels up or down.

It is a powerful environment for users to get to know and test their systems. It also provides (especially with StethoScope) a way to cautiously test and execute a program that may be controlling dangerous or fragile equipment.

The shell embeds an interpreted scripting language called TCL (the Tool Command Language). TCL is a simple-but-powerful language. It includes variables, control structures (if-then, while, for, etc.). With TCL, users can write simple scripts to, for instance, repeatedly execute a test sequence. Through the menu shell, virtually all the facilities of ControlShell are available to TCL programs.

5.5 Run-Time Tools

Finally, let us emphasize that RTI is a leading real-time tools vendor in many areas. As a result, ControlShell users benefit from a complete array of debugging, performance enhancing, and analysis tools.

First, ControlShell is fully integrated with RTI’s popular real-time data monitor, StethoScope. StethoScope allows users to monitor any variable in a ControlShell program, view it graphically, and store it to disk. Figure 12 shows a typical StethoScope session.

Figure 12 StethoScope

StethoScope provides on-line monitoring of any signal in a ControlShell system. Signals are displayed immediately as they change. The data can be stored for later analysis.
The state machine system has a tool called “LiveLook” that captures and animates state machine transitions. It highlights each transition and the destination state. You can capture sequences of transitions, play them back later and single-step through them. LiveLook is a powerful way to understand and debug your sequential logic.

RTI also has many other useful real-time tools, including an execution profiler, memory leak detection, and memory corruption detection.
6 Conclusions

We hope we've given you a taste for ControlShell. As you have seen, ControlShell is designed to be an application framework that enables the development of complex control and simulation systems. It offers a unique object model, integrated event and cyclic processing, an understandable run-time architecture, and sophisticated modal configuration management. It features superb run-time experimentation and debugging.

And remember...

- **ControlShell is the right solution for intelligent control.** It's specifically designed for building complex control systems. It works the way you think, letting you build well-structured, hierarchical systems easily. It directly attacks the sources of complexity that threaten your design.

- **ControlShell fits naturally into your team's process.** It lets you share work and reuse code through the magic of reusable components. It includes tools for all members of your team and phases of your project—from high-level design to final integration and test. It puts domain engineers in control without frustrating good programmers.

- **ControlShell accelerates your development.** Its fully-automated visual programming is fast and easy to use. Its specific structure and clean architecture gets your code running quickly. It includes powerful simulation, test, debug, on-line visualization, and debugging tools. ControlShell gets your real-world system running sooner.

In closing, let us assure you that RTI has a corporate commitment to treating customers as long-term partners. We want you to make a quality business decision, and get the tools and support you need to succeed. We stand by our products, our service, and our customers. You won't regret your partnership with RTI.

If you are developing a complex, possibly dangerous, difficult system, please consider all your options carefully. We're confident that you'll choose ControlShell.