2004-01-2478

Control System Architectures, Technologies, and Concepts for Near Term and Future Human Exploration of Space

Mr. Richard Boulanger
EASI, Inc.

Mr. David Overland
NASA Johnson Space Center

ABSTRACT

Technologies that facilitate the design and control of complex, hybrid, and resource-constrained systems are examined. This paper focuses on design methodologies, and system architectures, not on specific control methods that may be applied to life support subsystems. Honeywell and Boeing have estimated that 60-80% of the effort in developing complex control systems is software development, and only 20-40% is control system development [1]. It has also been shown that large software projects have failure rates of as high as 50-65% [2,3]. Concepts discussed include the Unified Modeling Language (UML) and design patterns with the goal of creating a self-improving, self-documenting system design process.

Successful architectures for control must not only facilitate hardware to software integration, but must also reconcile continuously changing software with much less frequently changing hardware [4]. These architectures rely on software modules or components to facilitate change. Architecting such systems for change leverages the interfaces between these modules or components.

INTRODUCTION

Life support systems are difficult to control at best. They are by nature complex, hybrid systems. Controlling the life support systems is not a simple task. Previous systems tests have been successful in only carefully monitored and controlled environments, very different from the conditions likely to be encountered during an actual space mission.

The computer data acquisition and control systems enabling future human exploration of space must support multiple missions and, therefore, support the evolution of both hardware and software technologies. A modular or component-based architecture for these control systems is indicated. Using a modular design to construct a control system doesn't make it adaptable, extensible or scalable, however. These factors must be designed in. An effective design process is critical to successfully designing and evolving such software systems.

Two design process techniques are discussed: design patterns and the Unified Modeling Language (UML). Design patterns capture and transfer knowledge specific to a particular problem area. Knowledge is captured in the form of templates that have been demonstrated successful to document and store this knowledge for new designers and programmers. The UML is a language to specify, construct, visualize, and document the components, their interactions, and how they are integrated.

Component-based development differs from object-oriented development in that a separate component and interface specification is developed. This means that intercomponent dependencies can be restricted to individual interfaces, rather than the whole component specification. The impact of change is reduced because one component may replace another as long as the specification includes the same interfaces.

A component may involve several interfaces; for example, a controller for an air processor subsystem could have a component that has interfaces with a sensor network, a database, and a human interface. New components are built to existing interfaces, with new functionality using existing interfaces or providing new ones. Component-based systems are an effective way to fast track the integration of research product into existing test or flight systems.

DESIGN PATTERNS

Experts tend to remember past models and designs, and often solve problems by reusing old solutions in the new context. In mature engineering disciplines, systems are commonly developed from these past models, experiences and designs. Software engineering is now to adopting architectural level patterns for code reuse. Key is the adoption of models that capture the design knowledge, facilitating early analysis of system properties. One way to capture this knowledge is the development and use of design patterns.

Design patterns originated with the work of building architect Christopher Alexander. Alexander refers to something he called "the quality without a name", which he describes as:

> *"To define this quality ... we must begin by understanding that every place is given its character by certain patterns of events that keep on happening there ... each building and each town is ultimately made out of these patterns ... and of nothing else. ... The more living patterns there are in a place ... the more it comes to life as an entirety, the more it glows, the more it has that self-maintaining fire which is the quality without a name."* [5]

Patterns do not make hard problems simple to solve. Patterns help people understand complex problems. They free people from repeatedly solving the easy problems. Patterns capture existing knowledge, allowing new solutions to be built on existing solutions.

Patterns are recorded in a format called the *Alexandrian Form*. A pattern is both a thing and instructions for making the thing. The form is particularly useful way to express patterns. The form looks like this:

- *The pattern name*: A name by which the pattern is called.
- *The problem* the pattern is trying to solve: This way people know when to apply the pattern to the problem they have.
- *Context*: A pattern may solve a problem in a particular context, but not make sense elsewhere.
- *Forces or tradeoffs*: Not all problems are clear. Forces clarify problem intricacies. Good patterns resolve one or more forces.
- *Solution*: Describes the structure, behavior, etc. of the solution, often detailing how to solve the problem.
- *Examples* are present in all good patterns.
- *Force resolution or resulting context*: a good pattern describes what it leaves unresolved or what other patterns must be applied.
- *Design rationale*: This describes where the pattern came from, why it works, and why it is used.

Pattern: People Know Best

Problem: How do you balance automation with human authority and responsibility?

Context: A highly reliable continuously operating system that tires to recover from all error conditions on its own.

Forces: People have a good subjective sense of the passage of time, how it relates to the probability of a serious failure, and how outages will be perceived by the customer. The system is set up to recover from failure cases. People feel the need to intervene. Most system errors can be traced to human error.

Solution: assume that people know best, particularly the maintenance folks. Design a system to allow knowledgeable users to override automatic controls.

Resulting Context: People feel empowered, however they are also held accountable for their actions. People feel a need to intervene. There is no perfect solution to this problem, and the pattern cannot resolve all the forces well. *Fool Me Once*[1] provides a partial solution in that it doesn't give humans a chance to intervene.

Rationale: Consider the input command to unconditionally restore a unit. What does "unconditional" mean? Let's say the system thinks the unit is powered down, what should happen when the operator asks for the unit to be restored unconditionally? Answer: try to restore it anyhow, no excuses allowed, the fault detection hardware can always detect the powered-down condition and generate an interrupt for the unit out of service. Why might the operator want to do this? Because the problem with the unit may be not with the power but with the sensor that wrongly reports that the power is off.

Author: Robert Gamoke, ATT Bell Laboratories, 1997

Figure 1. An example design pattern.

Design patterns have been applied to fault-tolerant telecommunication systems. Two unique characteristics of telecommunications software that coincide with Advanced Life Support control software are reliability and human factors, making them good candidate patterns for further investigation. One of these patterns is presented as an example in Figure 1, above. [6]

Patterns are *not,* however, a silver bullet. Most patterns affect software design, with no quantifiable effect on lines of code. Patterns capture existing, well-proven design experience. Patterns *should* look familiar, especially to senior programmers and designers. A written pattern gives a non-expert designer, someone

[1] *Fool Me Once* is a pattern which limits the ability of an application to cause a system restart to a single occurrence.

new to the control of ALS systems, some reusable building blocks for software development. Patterns capture the most important information that must be passed on to the next generation of designers and programmers.

THE UNIFIED MODELING LANGUAGE

The Unified Modeling Language (UML) is the industry standard for modeling and documenting object-oriented systems. It describes a standard set of notations and diagrams that describe object-oriented systems and it describes what those symbols mean. The UML is a process by which a designer (e.g. a controls system developer) and a customer (e.g. a life support system PI) can reach mutual understanding of how a subsystem is designed to operate, nominally, off-nominally and failure modes. Application of the UML to a control system design problem begins a dialog between software and hardware designers, the result of which peels back the layers of knowledge the hardware designer possesses, and allows that knowledge to be documented in a clear, concise manner which control systems programmer and designers can understand and translate into effective code.

Using the UML to develop of a control system produces several artifacts. These artifacts describe how the control system operates, how humans and other parts of the systems interact with control system, how the system's various components interact, and what components are required. These artifacts are called, respectively, the Concept Model, Use Case Models, Component Specification, and the Components. Artifacts are developed iteratively, as shown in figure 2, below.

Creation of the specification artifacts is tricky since various artifacts have clear dependencies. Workflow tasks can be summarized into three stages: component identification, component interaction, and component specification. In all of the stages, component refers to both the component and its interface(s). Figure 3, right, shows the process, interactions, product inputs and artifact outputs. The process is iterative: each of the processes shown is executed concurrently.

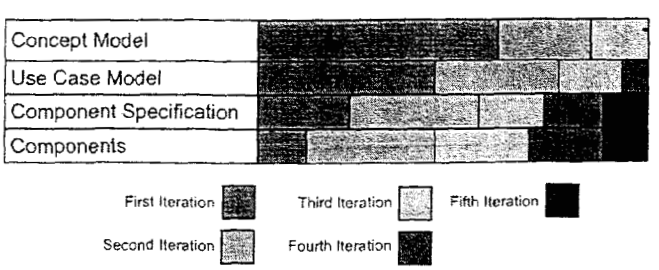

Figure 2. UML artifact development is an iterative process.

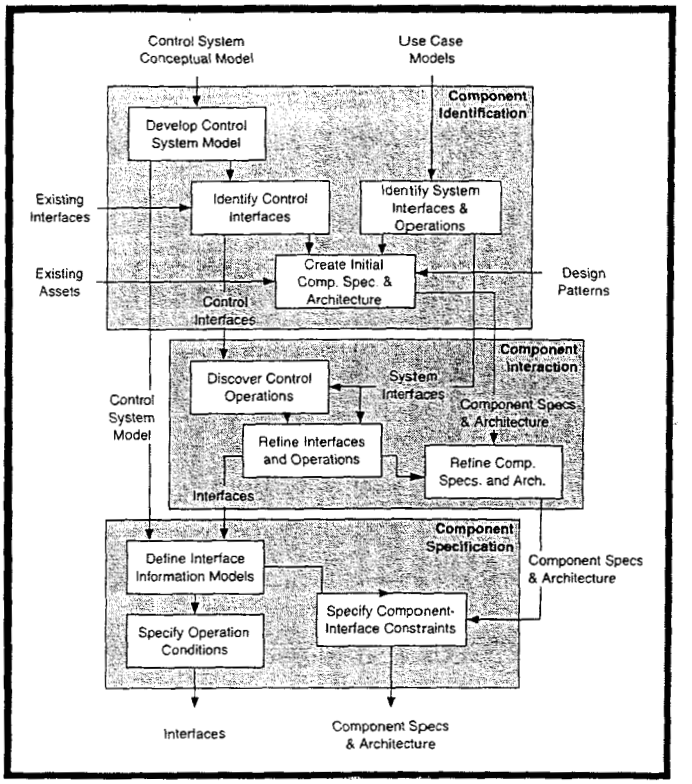

Figure 3. Workflow diagram for developing component-based control systems.

COMPONENT IDENTIFICATION – The component identification stage takes as input the control system conceptual model and use case models. The goal of this part of the process is to identify the initial components and set of interfaces for the control system. Together with design patterns and existing assets, (e.g. communication standards, databases, hardware, etc.) these are then used to construct the initial control system. The initial cut at the components can be very broad at this point. They will be refined as the process continues and iterates. The operations that the control system must perform should be identified at this point, which defines the initial interfaces the system will require.

COMPONENT INTERACTION – The component interaction stage uses interaction models to discover control system operation. As more interactions are considered, common operational modes and usage patterns will emerge. Operations can be moved from one component to another, clarifying the interface responsibility. The goal here is to minimize the dependencies between component objects. Details of the control system structure emerge during this process, with a clear understanding of the interaction between components.

COMPONENT SPECIFICATION – The final stage is component specification. At this point, detailed operations and constraints are defined. What conditions must be exist prior to and after each component

operation are defined. This helps determine missing parameters, missing information, wrong assumptions, etc.

UML MODELING TECHNIQUES – The UML utilizes several notations to diagram the artifacts that it produces. The UML uses the term *model* to refer to specific types of diagrams that represent specific system abstractions. These diagrams represent concepts of how information flows though a system, how users interact with the system, and how components interact with each other. There are three basic types of diagrams for modeling component-based control systems using the UML: use-case diagrams, class diagrams and collaboration diagrams.

CONCEPT MODEL – The control system conceptual model is not a model of software, but rather a model of the information that exists in the problem domain, i.e. a model of the process by which a life support system operates. The purpose of the conceptual model is to capture concepts and define relationships. These concepts and relationships are then used to define the components necessary to adequately and completely control the system in any state.

CLASS DIAGRAMS – Class diagrams describe the static structure of a system. Classes are depicted as rectangles divided into compartments for name, attributes, and operations. The relationship between classes is shown using associations and cardinality. Associations are lines from class rectangle to class rectangle that show there is a static relationship between the two classes. Cardinality is a notation that shows multiplicity between cases. For example, figure 4 shows a class that represents the information that might be contained in a database of process control data.

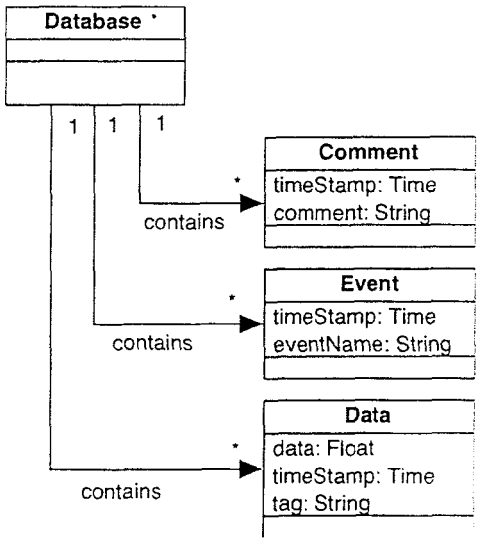

Figure 4. Example class diagram.

USE CASE DIAGRAMS – Use case diagrams model how the user interacts with the system. The participants in a use case are *actors* and the system. An actor is an entity that interacts with the system, typically a user. An actor can be another system, but, if it is, the details of that system are hidden — we simply see a predictable "person". One actor is always defined as the initiator. The other actors (if any) are used by the system to meet the initiator's goal. In a use case, the actors interact with the system as a whole, not some part of it. It is also important to note that the method the actor uses to communicate with the system is of no concern, whether it is a graphic user interface, an Internet connection, or a control panel button.

It should also be pointed out that the purpose of the use case is to meet the immediate goal of the actor. This leads to a commonly occurring problem with use case diagrams: scope and size. There is no consensus on the issue; however, a first approximation is to make the use case smaller than the conceptual model process but larger than a single component operation. The purpose of the use case is to meet the immediate goal of the actor, be it changing a control set point or checking material stores. It will include everything that can be done now by the system to meet the goal.

COLLABORATION DIAGRAMS – The rectangles in these diagrams represent component objects that support the interfaces that are shown. Since it is possible (even likely) for components to support more than one interface, there may be two or more rectangle for some components. The links are instances of the interfaces shown unless they are transient: existing only as long as the collaboration between the two components occurs.

ARCHITECTURES

The control system architecture is the principal means by which change to a system is effected. Modern control system architectures are made up of networked sensors, actuators and software components, at various levels, which, together, effect change in that system. Regardless of complexity, the basic system consists of a sensed value, an algorithm for determining sensed value goodness, and an actuator for changing the system state based on that measure of goodness.

Control system architectures are typically hierarchical or heterarchical. Hierarchical systems are characterized by a controller that is located at the top of a hierarchy of sensors, actuators and passive algorithms. Sensor input data flows up the hierarchy to the controller, which issues commands that flow down to the actuators. A network of sensors and actuators that cooperate to achieve system wide goals characterizes heterarchical systems.

Control system software for advanced life support is distributed across the processors and subsystems that comprise the system. Currently, the developer of each physico-chemical or biological processor is responsible for developing the software needed to run her equipment. In the near future, this process will have to change. To give a control system the highest degree of flexibility with which to succeed, it needs to have access to all the sensed inputs and actuators that are available to it. The control system software needs to be able to detect a condition by reaching down, into a hardware designer's equipment, diagnose that condition, and then reach down again, possibly into another designer's equipment, and effect the change necessary to correct the situation. The process by which equipment is supplied to NASA today does not allow this to occur.

The ability to change and evolve must be designed into the software we use to control the life support hardware. Failing that, there must be a cohesive, effective way to encapsulate these subsystem controllers, allowing access to their data and actuators until there is such a standard interface.

COMPONENT-BASED

Component-based control systems utilize interactive software abstractions called components. These abstractions possess three attributes: stored knowledge or data, the ability to process knowledge or function, and the means to share knowledge or interfaces. By judiciously determining the interfaces which components share, component dependencies are created. These dependencies determine to a large extent the type and number of components required by the system. Definition of a common interface and building *to* that interface allows future work to increase functionality, while maintaining backwards compatibility. Components can be extended to include additional functionality as long as the interface's original functionality is left intact. Components are the basic building blocks of modern, extensible, evolvable control systems.

RECONFIGURABLE

A reconfigurable control system is a component-base control system where the components can be changed in real-time. For example, a model-based controller with a relatively large time step is controlling a water processor. The large time step is used to reduce computer overhead in situations where the model predictive accuracy is not that critical. However, if sensors indicate, a more accurate (and more computer intensive) model could be substituted on the fly, adjusting is a more controlled manner the trajectory of the water processor.

The ability to reconfigure a control system in real-time provides a valuable tool for integrating new control technologies as well as proven alternatives into the control mix. Imagine a system that is disparate, comprised of scattered processors, simulations and controllers. Imagine that this conglomeration of hardware and software shares a common set of interfaces and that components of the system can be switched in and out at will. The system can be configured to run a ground-based simulation of an advanced life support mission or the system could *be* the mission itself, with ground expertise, simulations and alternate control strategies to the spaceflight hardware.

CONCEPTS

It is likely that future spaceflight hardware and software will utilize separate spiral development cycles. This will inevitably lead to a circle-spiral process model, with stable systems being represented by circles, and iterative development proceeding along spiral paths approaching those circles. The stable configurations can be software release levels, architectural frameworks, or hardware configurations. One problem with designing a process for integration is *doing the hard part first.* The evolutionary nature of software implies that, given this heuristic, the hard parts of the software should be designed *first.* Given what is know about schedule risks and software, this may seem obvious to the casual observer, but it is often difficult given pressures to produce interface demonstrations and so forth. Reliable control of advanced life support processes, therefore, needs to be effectively demonstrated prior to creation of sharp looking interfaces. Given that a reliable control demonstration is the first order of business, suitable hardware or a hardware analog must be available for control system software to succeed.

Two mechanisms for management of change are to implement improvements in the form of large portions of the system, so-called large lump development, or to implement smaller, continuous growth, or so-called piecemeal growth. Large lump development is relatively static, with change often occurring in leaps based on external influences to the system. Piecemeal growth is more dynamic, typically in response to internal system influences. Both mechanisms have a place in control systems design. Implementation of new technologies that fundamentally change the way a control system operates is an example of large lump development. Whereas piecemeal growth may be as simple as developing a replacement component that utilizes a new network interface or service.

Management of change is the most important priority to any long-term software project. Successfully managing change reduces risk of project schedule slip or failure. It also quantifies schedule progress, allowing accurate assessment of effort required.

CONCLUSIONS

In 2001, an estimate was made of the size of the facility control system for a human-rated test bed call the BIO-Plex. The estimate was based on the assumptions that a component-based control system would be used and that no "high-level" or autonomous control was part of the facility control system. The estimate was on the order of 10,000 function points. Projects of this size have failure rates on the order of 50 percent.

Since that estimate was made, the test facility has grown in scope, to include more diverse elements such as human factors, food processing, and advanced environmental monitoring and control. The probability of success for the control system software design and implementation has not decreased, however.

Only by successfully managing the design for change will these efforts succeed. So we stand at a crossroads: do we design using the technology of today, the same technology that brought us to the moon before, or do we incorporate new discoveries, new technologies as they are proven and made available? The choice is ours to make. Where one path supports a human presence in space, the other will leave only leave footprints.

REFERENCES

1. Honeywell, Boeing reference
2. Jones, Capers T., *"Estimating Software Costs"*, McGraw-Hill, 1998. (ISBN 0-07-913094-1, 724 pages)
3. Boulanger, R., D. Overland and H. Jones "Evaluation of Fieldbus and Software Component Technologies for Use with Advanced Life Support", 31st Intl. Conf. On Environmental Systems (ICES), SAE Technical Paper No. 2001-01-2299.
4. Rechtin, Eberhardt and Mark W. Maier, "*The Art of Systems Architecting*", CRC Press, LLC, 1997. (ISBN 0-8493-7836-2, 266 pages)
5. Alexander, Christopher, "*The Timeless Way of Building*", Oxford University Press, 1979. (ISBN 0195024028, 568 pages)
6. Jones, Capers T., "*Software Assessments, Benchmarks, and Best Practices*", Addison-Wesley, 2000. (ISBN 0-201-48542-7, 657 pages)
7. Rising, Linda, ed., "*The Patterns Handbook: Techniques, Strategies, and Applications*", Cambridge University Press, 1998. (ISBN 0-521-64818-1, 549 pages)
8. Cheesman, John and John Daniels, "*UML Components: A Simple Process for Specifying Component-Based Software*", Addison-Wesley, 2001. (ISBN 0-201-70851-5, 176 pages)

CONTACT

Mr. Richard Boulanger
EASI
NASA Ames Research Center
M/S 239-8
Moffett Field CA 94035-1000
Voice: 650.604.1418
Facsimile: 650-604-1092
rboulanger@mail.arc.nasa.gov

Mr. David Overland
NASA Johnson Space Center
Mail Code ER2
Houston TX 77058
Voice: 281.483.4304
Facsimile: 281.483.3204
david.overland-1@nasa.gov