Real-Time and Object-Oriented Issues for an Inspection Workstation Application

John Albert Horst
100 Bureau Drive, Mail Stop 8230
The National Institute of Standards and Technology (NIST)
Gaithersburg, Maryland, USA 20899-8230
john.horst@nist.gov
voice: (301)975-3430

Abstract

We describe a real-time component-based system for an inspection application. We chose the inspection application and the accompanying task (or scenario) so that we might fully exercise and test our theories about real-time complex systems, system architectures, design methodologies, and software tools. We will describe the application, give a history and description of our system architecture and design methodology, describe the real-time software tools we used, and conclude with a discussion of real-time and object-oriented issues we faced.

1 The Inspection Application

Our inspection workstation consists of a coordinate measuring machine (CMM), an analog 3D contact probe, a charge-coupled device (CCD) camera with frame grabber, and control computers. The CMM is Cartesian in the sense that axis motion and axis position sensing are along the three orthogonal axes. The contact probe and camera are mounted on the CMM arm. The software controller sends velocity commands to each of the three axis motors every 5 ms and it reads each of three axis positions every 2 ms. The axis velocity commands are converted to voltages by a digital-to-analog converter. The voltages drive the motors. Figure 1 shows the CMM arm, the part to be measured, the camera mounted on the arm, and the analog contact probe. The application is more fully described in [Messina 99].

The application performs the following scenario. The operator specifies the features that need to be verified by measurement. An inspection plan is generated automatically from the computer-aided design (CAD) solid model of the part. The plan is translated into dimensional measurement interface standard (DMIS[1]) code. A DMIS interpreter [Kramer 99] converts the DMIS code into canonical control commands for the CMM and vision subsystems. The CMM is commanded to move to a predefined "bird's eye view" position. The part to be measured is placed

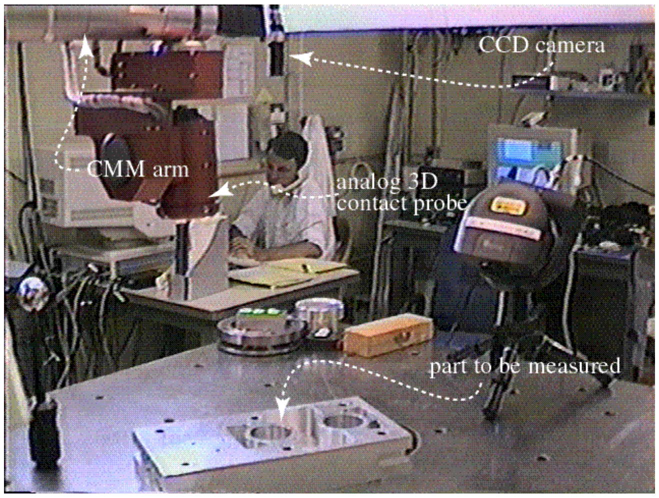

Figure 1: The inspection workstation.

[1] DMIS is approved by the American National Standards Institute (ANSI) as a vendor-neutral language for part inspection programming.

on the CMM table in an arbitrary position and orientation. The human operator signals that the part is on the table. The system determines the position and orientation (i.e., pose) of the part using the camera and computer vision algorithms. The inspection plan is performed.

The application and scenario were crafted to have the following characteristics (in order to exercise our theories adequately):

- large-scale, complex system requiring modularization and encapsulation for system perspicuity
- rich sensory processing and world modeling
- real-time sensing and command
- on-line task planning
- processing and handling of errors
- integrated human operator and human developer interaction
- distributed control over multiple target operating system and processor platforms
- simulation and animation
- rich interaction between CAD model features and sensed features

Our system contains on-line inspection plan generation, integrated operator interface, the use of multiple software tools, the control of multiple pieces of sensing and actuation hardware (*e.g.*, probe, CMM arm, CCD camera.), and the execution of all on multiple computing platforms. We used a Power PC™ (PPC)/VxWorks™ target for real-time control, a Sparc™/Solaris™ target for the vision subsystem, and two more separate Sparc/Solaris platforms for inspection planning and plan execution[2]. Simulation of inspection results and animation of CMM motion are displayed on any system with Virtual Reality Modeling Language (VRML) software. A simulation of the CMM runs on a Silicon Graphics, Inc. (SGI) computer.

Rich interaction with the CAD model occurs in both the inspection plan generation and the pose estimation tasks. The pose estimation task requires a computer vision subsystem that consists of image processing (the generation of constant curvature arcs and line segments), computer vision (the matching of sensed feature set parameters with model feature set parameters), and operator interface (to handle errors and signal a “good” pose). A rich and complex world model is also required. CAD solid model features are converted to model feature parameter sets. The model features are ordered off-line in preparation for on-line matching with sensed feature set parameters.

Simulation and animation occur at several levels. The system design for CMM motion and probe control can be linked into several separate executables that have varying types and degrees of simulation, from all simulation to no simulation. This has arguably improved debugging and testing efficiency. Simulation and animation are also provided by two separate software subsystems that monitored motion and measurement data and provided real-time motion simulation/animation of the CMM.

Real-time response was required for both motion control and measurement via contact probe sensing. State-of-the-art CMM motion speeds are upwards to 300 mm/s and higher. Additionally, real-time requirements on measurement are even stricter. Without real-time, dependable control of CMM arm motion, expensive contact probes can be damaged (*i.e.*, we must stop predictably when we touch the part).

The component-based control system design and run-time software tools we used provided for rich human developer interaction as well as human operator interaction.

2 The RCS architecture

The system architecture used is the NIST Real-time Control System (RCS) [Albus, 96]. RCS specifies a generic building block (or template control node) that is copied throughout the system. A conceptual view of an RCS generic building block is illustrated in Figure 2. The control nodes are connected according to the rules established by the architecture. Each control node contains modules with appropriate taxonomy. RCS does not require that the modules within a control node map directly to processing modules, though in our implementation, they do. The taxonomy is an attempt to divide the labor of a control node (a building block) into subcomponents and interconnections that will minimize component-to-component communications bandwidth, provide for component reuse, and minimize component complexity. The intra-component modules within each node are sensory processing, world modeling, value judgement, and behavior generation (as shown in Figure 2). Here are some examples of what is commonly performed within these modules. Plan generation and execution are done in

[2] Reference to specific commercial vendors and their products does not imply endorsement by NIST

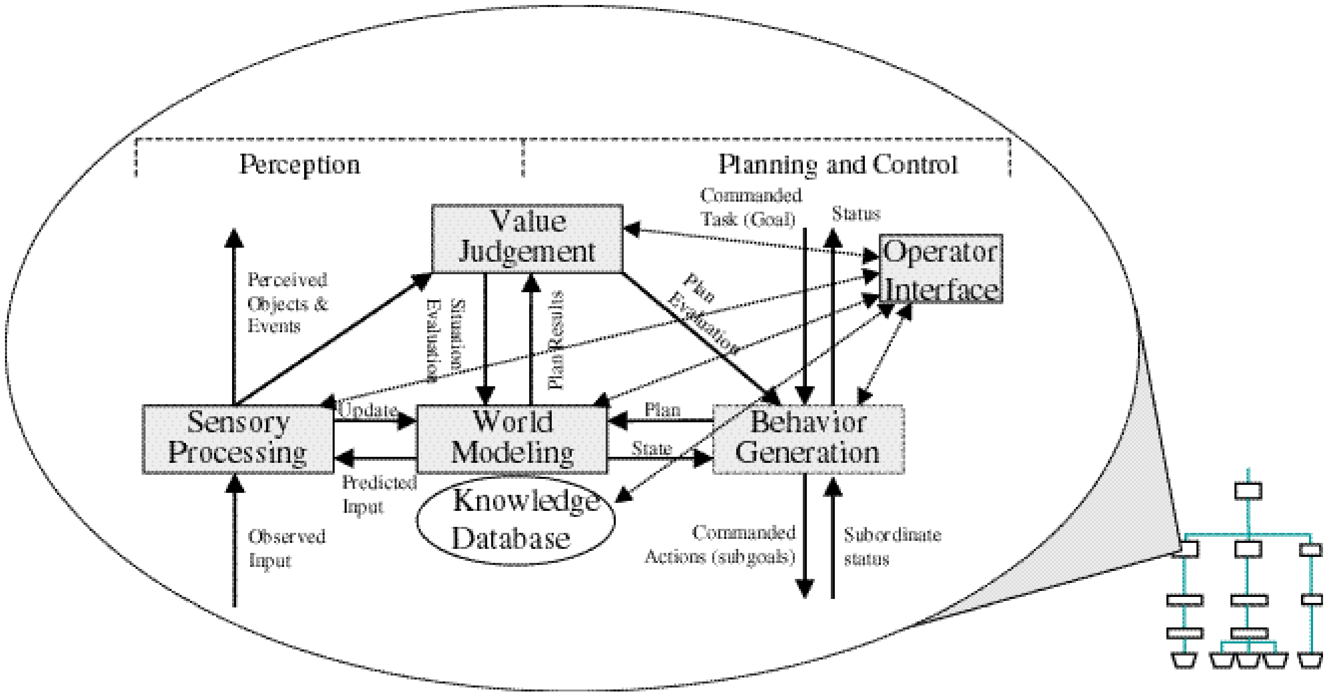

Figure 2: The RCS generic building block (or control node).

behavior generation, image processing is done in sensory processing, pose estimation is done in world modeling, and model feature set attributes for the part to be measured live within the knowledge database. This basic pattern of the node is copied throughout the system, but each node varies in temporal and spatial scope depending on where it lives in the hierarchy. Of course, the actual node contents also vary widely. This is roughly equivalent to human military hierarchies where, for example, a general is concerned with plans and actions months in advance and entire battalions of soldiers over many battlefields, but the foot soldier may be concerned with plans and actions for only a few minutes over a small area.

The number and placement of control nodes in the system hierarchy are based on the tasks to be performed and the actuators that have to be controlled, which is to say the hierarchy is generated by both top-down and bottom-up considerations. It is also an iterative process [Quintero 92]. As the system grows and develops, one may discover a need to add or subtract nodes, levels, or branches in the hierarchy. The number of hierarchical levels in the system is generally determined by a trade-off between system complexity and system overhead. Several other guidelines help determine the number of levels including coordinate frames of reference and the type of sensor data processed [Albus 96]. An example of the latter dictates the number of levels in our vision subsystem. The lowest level (servo) handles the pixels, the next highest level (prim) groups pixels into linear features (line segments and constant curvature arcs), and the highest level (emove) forms linear features into feature groups or patches. For motion control applications (like this one), three levels, elemental move (emove), primitive (prim), and servo, seem to be sufficient to execute high level motion commands, like `CMM_traverse_emove` (see Figure 4).

Control nodes have a standard and a non-standard interface. The standard interface is between supervisor and subordinate nodes. This interface always consists of command from supervisor to subordinate and status from subordinate to supervisor. The non-standard interface allows any node to communicate with any other as required. For example, in our application, we provided the `probe_touched` event to several nodes at various locations in the hierarchy. Finally, a node is allowed only one supervisor node per sampling cycle.

A description of the RCS methodology will further clarify these concepts.

3 The RCS Methodology

The RCS methodology consists of step-by-step instructions for building a complex, real-time system. The goal of the methodology is to facilitate system design and maintenance efficiency

To begin, the system developer defines the highest level task and identifies the resources available (e.g., sensors and actuators). For illustrative purposes, we'll examine two mid-level tasks, `inspect_part` and `init`, used in our inspection application. Our resources are the CMM, the CCD camera, and the probe, as well as computing platforms, for both hard real-time and soft real-time performance.

Based on the node placement and interconnection guidelines of section 2, the developer "decomposes" the high level tasks into subtasks as illustrated in Figure 3.

These tasks are then grouped into controllers based on the bottom-up analysis of actuators to be controlled. We have a probe, a camera, and a CMM arm to control. Therefore, we have three branches in our hierarchy. The grouping of tasks into nodes for our example task is depicted in Figure 4.

The next step is to create finite state machines (FSM) for each of these commands at each of the control nodes. These FSMs together define system behavior. An example of an FSM for a prim level `goTo` task is found in Figure 5.

The final step is to map the nodes onto specific computing platforms. For example, the vision branch in Figure 4 is mapped onto a soft real-time platform (Sparc/Solaris) and the CMM and probe branches are mapped onto a hard real-time platform (PPC/VxWorks).

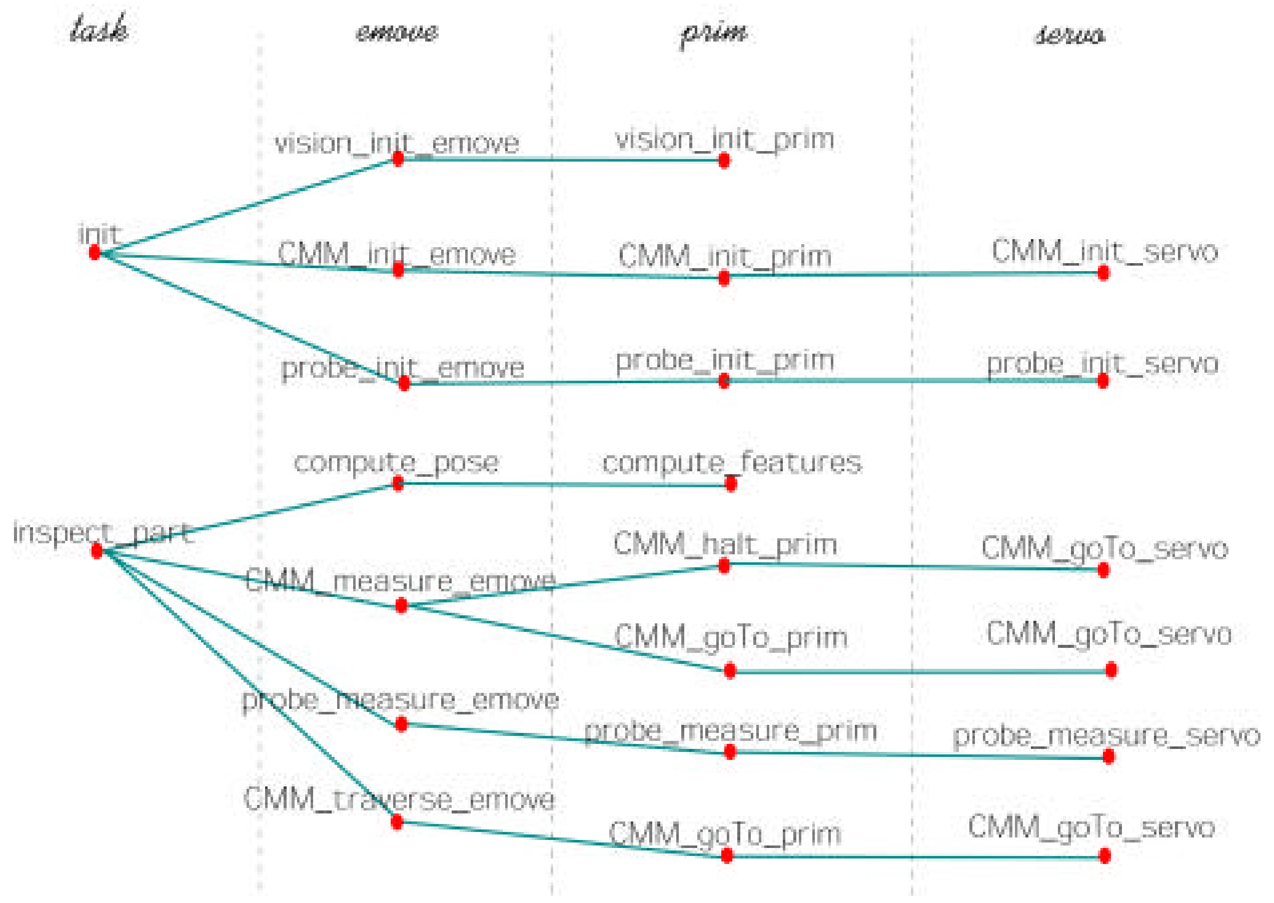

Figure 3: A task tree (task decomposition) for the `inspect_part` and `init` tasks.

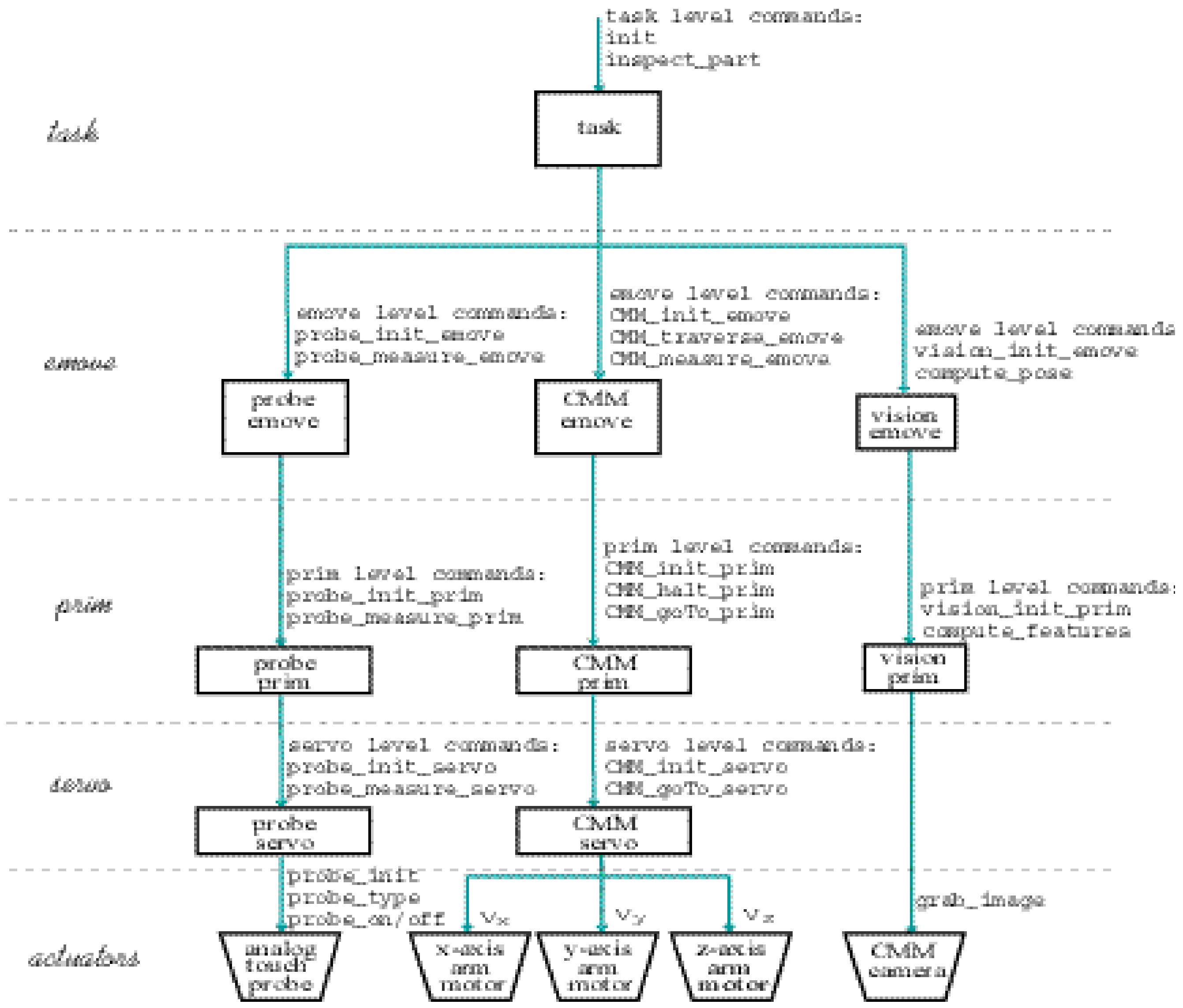

Figure 4: A hierarchy of control nodes (RCS building blocks from Figure 1) with tasks mapped into nodes for the `inspect_part` task.

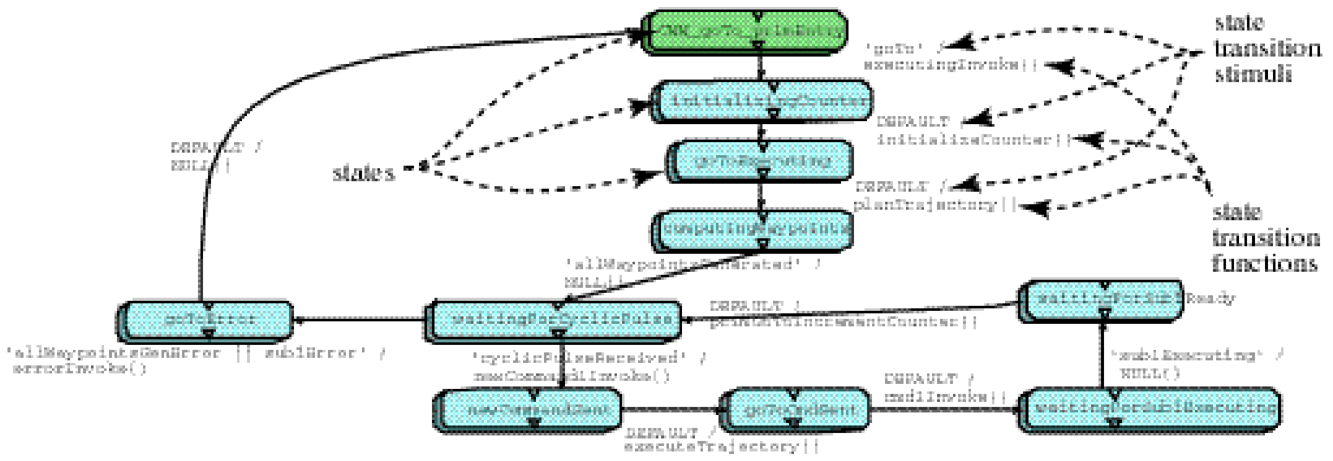

Figure 5: A finite state machine diagram (generated in ControlShell) for the prim level `CMM_goTo_prim` task from Figures 3 and 4. Note the use of the cyclic pulse stimulus sent from the synchronous process.

4 Tools to support RCS

The RCS architecture and methodology need tool support to facilitate system design and maintenance. For this application, we use two tool sets. For distributed communications we are using the Communication Management System. This tool and other supporting tools under development at NIST form a comprehensive tool set for RCS style system development [Shackleford 99]. This development system was used at higher levels in the application system hierarchy (task level and above).

For the system hierarchy of Figure 4, we are using a commercially available tool, called ControlShell™, from Real-Time Innovations, Inc. (RTI). We will focus our discussion on the ControlShell tool set. ControlShell is actually not a single tool, but a set of several integrated software tools that can be used to develop large and complex control systems. It is a graphical, component-based tool set for object-oriented, real-time system development allowing synchronous and asynchronous execution for a variety of operating systems and target hardware. The target application domain for the tool set is electromechanical systems, but it is not inherently limited to that domain.

ControlShell has a diagram editor in which the user develops a graphical design for the application. The diagram editor allows definition and graphical interconnection of components. Components requiring synchronous (cyclic) execution of code can be grouped into sampling environments for execution. FSM components are also graphically defined in the diagram editor and are mapped into an asynchronous process for execution. At lower levels in the RCS hierarchy, we sometimes need to run portions of an FSM on a cyclic clock. Sending a cyclic pulse from a cyclically executing component to cause an event stimulus in an FSM satisfies this need.

There is a one-to-one map between the graphical design and what executes, *i.e.*, what you see is what executes (WYSIWE). Within a single graphical design, one can link component subsets to several executable systems, *e.g.*, one can define both simulation and real systems within the one design.

Component-to-component interface components can be defined in the diagram editor. These interfaces encapsulate user-defined method calls and data.

A run-time shell provides an execution environment for the application within the host operating system (VxWorks, Solaris, etc.). It allows the execution of compiled code, the modification of data values (at run-time), debugging, and other facilities. The compiled code is a relocatable object and, therefore cannot execute without the run-time shell tool. Data modification without recompile is available due to a run-time data binding facility that dynamically binds all data to the compiled code each sampling cycle.

A software oscilloscope, called Stethoscope™, is available for viewing any numerical data defined in the data dictionary, without interfering with the real-time process

A repository facility allows component and application reuse within and among team members.

RTI also offers several other tools that integrate with the ControlShell tool set. NDDS™ is used for distributed, real-time communications. ScopeProfile™ is used for real-time process analysis. MemScope™ is used for analyzing and debugging memory problems. None of these tools were used.

5 Developing a new RCS application with ControlShell

We have developed a template system in the ControlShell environment that will facilitate RCS style real-time system development. This template system is a ControlShell executable consisting of various components that can be used as a template for creating a new RCS-based application. To develop a new application or a new branch in an existing application, the user simply copies and edits the template system files. The template system consists of one branch of three RCS nodes as shown in Figure 6. The template system also contains reusable FSMs for `init` and `halt` commands. To add additional commands (tasks) to nodes, the developer would make a copy of the `init` FSM component, edit it as required, then add it to the parent FSM component The template parent FSM is shown in Figure 7. The newly generated command would also have to be added to the appropriate command interface.

In the template system, generic sensory processing and world modeling components are merely stubs to which application-specific code would be added as needed and compiled. Template interface components for command, status, sensory processing (SP), and world modeling (WM), as well as intra-node interfaces such as SP to WM interfaces, have been defined in the template system. The generic template component in ControlShell, implementing the RCS building block of Figure 1, is shown in Figure 8. This component is the internals of the "node" component of Figure 6. Within the "BG_COG" component of Figure 8 is the parent FSM component of Figure 7.

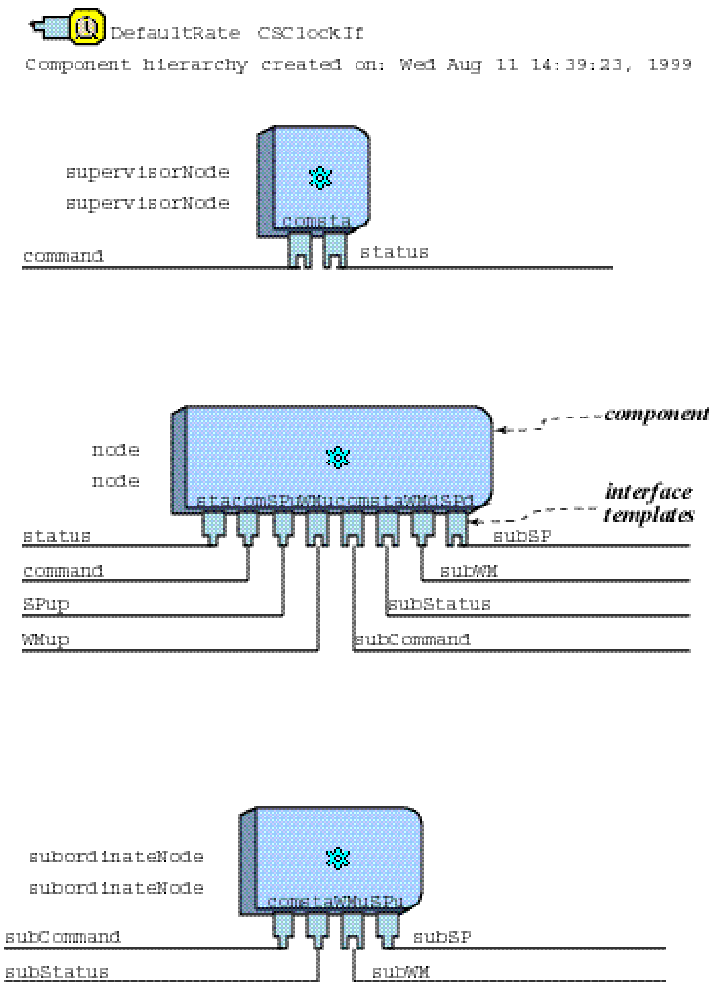

Figure 6: The generic template system for facilitating RCS designs in ControlShell.

6 Real-time Issues

The inspection application and scenario were chosen, in part, due to the real-time and distributed control challenges we would have to overcome. Both RCS and ControlShell have unique real-time issues. The integration of the two technologies was the stimulus for some important real-time effects.

Processing models for RCS have typically handled real-time by specifying that the control nodes, which execute FSMs, are required to 1) have deterministic, non-blocking execution and 2) execute, worst case, in less than one cycle period. Other RCS execution models require cyclically executing FSMs [Quintero 92]. While helping assure determinism, this system overly constrained certain aspects of execution. For instance, if the nodes were executed each cycle on one processor and sequentially from the top to the bottom, a high level command would reach the bottom node in one cycle. However, status would take *n*-1 cycles to reach the top node from the bottom node for an *n* level system, because of the top-down node execution ordering. Additionally, a node could not pass through many states per cycle. While real-time efficient performance can still be met with these constraints, system perspicuity is sacrificed, since for the sake of clarity, it is often helpful to

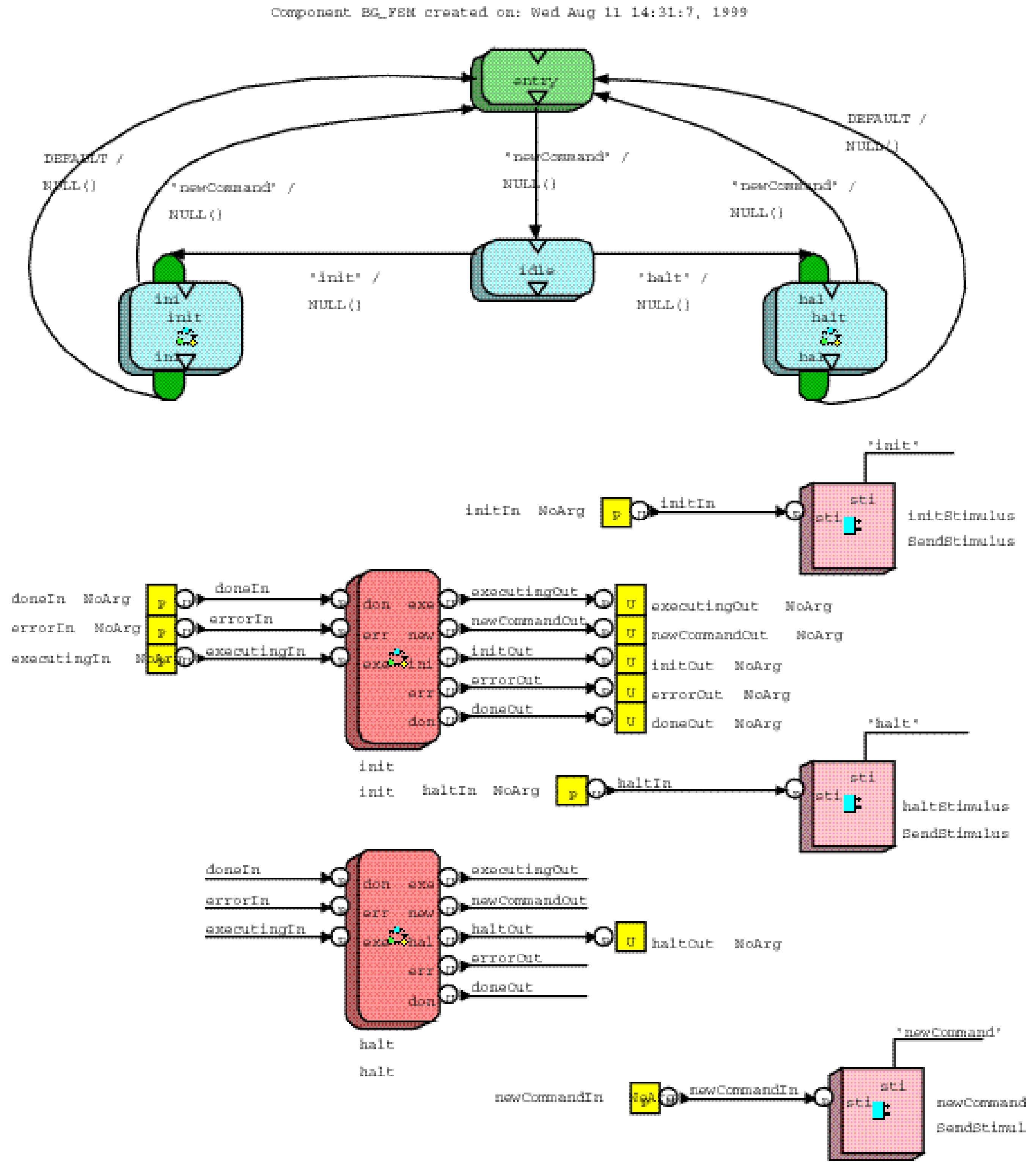

Figure 7: The generic template parent FSM component (in ControlShell) for an RCS node. Note that the `init` and `halt` FSMs are encapsulated within this parent FSM.

define several states with minimal or no processing per transition. For example, such a situation occurs between the states, `initializeCounter` and `computingWaypoints`, in Figure 5, since no stimulus is required for transition. In the alternate execution model, there are no asynchronous processes in the real-time execution system. According to this model, adding interrupts will sacrifice determinism, a key element of dependable systems. However, in an execution model like ControlShell, we have both synchronous and asynchronous processes at our disposal. Each process executes as a separate process in the real-time operating system, but is intertwined through method calls and shared data in the RCS design. Such a link between synchronous and asynchronous processes has at least two beneficial effects:

- since we model commands and status as method calls, the method calls are asynchronous, avoiding the *n*-1 delay mentioned earlier
- since the FSMs are asynchronous, if there is sufficient processing time during a given cycle, the system can process as many stimuli and state transitions for which there is sufficient processing power

Finally, we found that successful real-time execution was only possible when we gave a higher priority to the synchronous process than that given to the asynchronous process. This is, in part, because we must guarantee that the tasks of the asynchronous process never cause the tasks of the synchronous process to fail to complete in any sampling cycle. The asynchronous process is roughly equivalent to a background process for the system, which we execute with processor time remaining after execution of the cyclic modules. Therefore, our processing model for RCS still requires that we have deterministic, non-blocking execution of the synchronous code and that code must always execute within the sampling period of the sampling loop. However, under the new processing model, we have the freedom to put FSMs in the asynchronous process, which gives two benefits (without seeming to sacrifice real-time, dependable performance):

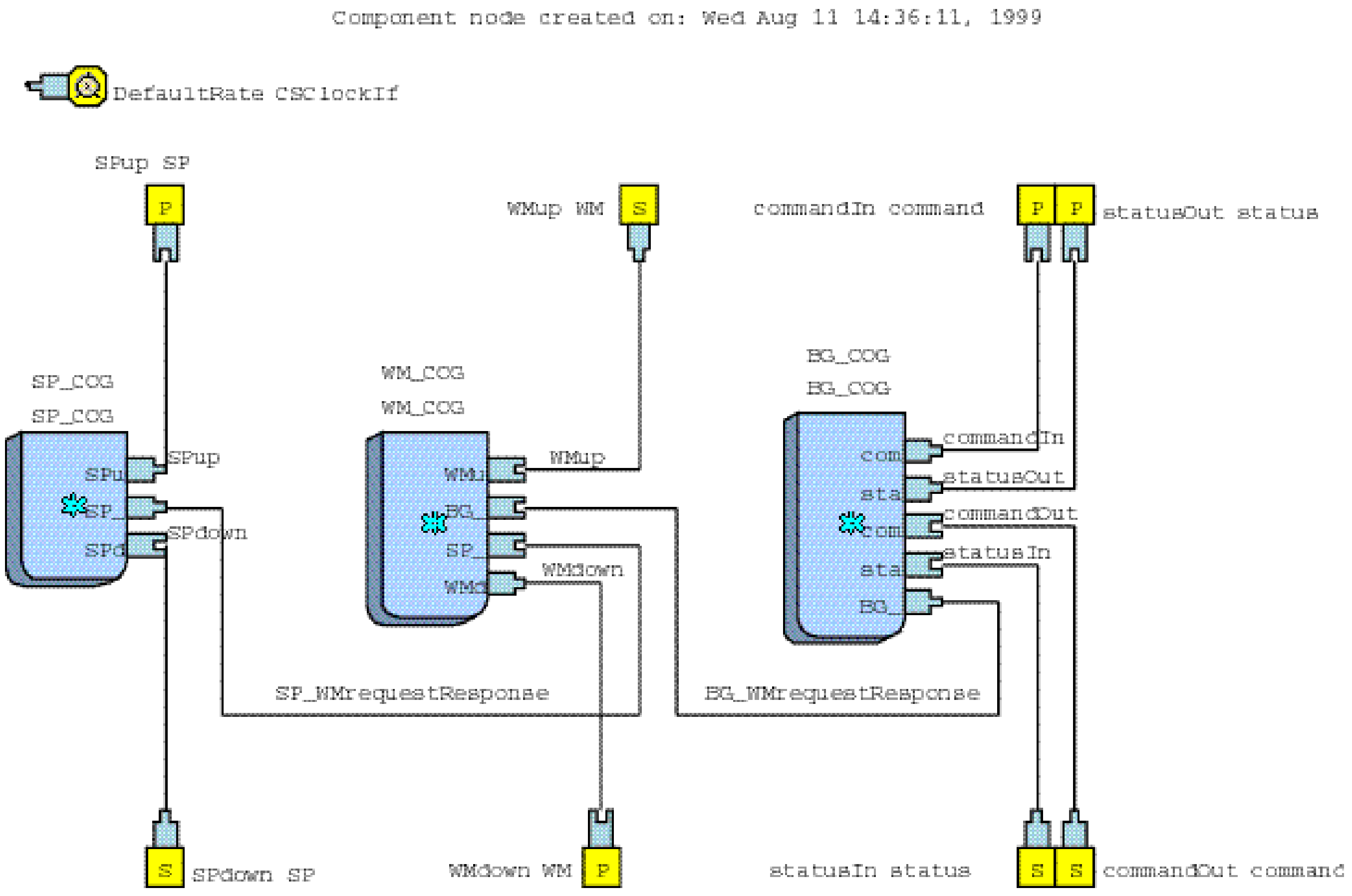

Figure 8: The generic template component for an RCS node. This is our implementation of the RCS building block of Figure 1 using ControlShell.

- the ability to design finite state machines so that nodes can transition through multiple states in a single cycle
- more efficient processor usage

7 Object-oriented issues

In the software industry, there are many and varied uses of the terms, architecture, components, and objects: We will simply describe how we have defined and used them and, more importantly, discuss how they interact in our system.

RCS has been shown to map successfully into an object-oriented environment [Huang 96]. Our work here is to make this claim manifest in a real application with a commercial off-the-shelf (COTS) component-based objected-oriented tool.

In our view, objects support components, components support the architecture, components support objects, and the architecture supports components. To be of any value, this support interaction, as suggested in Figure 9, must help us reach the goal of software engineering, namely, to discover and create theories, architectures, methodologies, and tools that facilitate the software lifecycle.

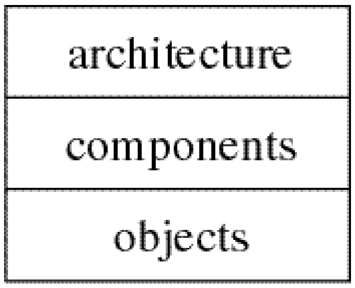

Figure 9: Simplified interdependency among architecture, components, and objects

The ControlShell tool defines the nature of the interface between components and objects in the tool and RCS defines the interface between the architecture and the components. We will now examine how our system can be viewed from the architectural, component-based, and object-oriented perspectives, in turn.

From an architectural perspective, our system

- defines component boundaries carefully to minimize data bandwidth between components, facilitate reuse, and keep any complexity to a minimum
- defines building block template components that can be used to facilitate design
- defines component interfaces and handshaking between control nodes
- defines components within a control node and the interfaces between those intra-node components (see Figure 2)
- defines component taxonomy

From a component-based perspective, our system

- can encapsulate other components and objects, therefore, components do not have to map to a specific class as do objects, *i.e.*, components provide further encapsulation to the system
- supports “what you see is what executes” (WYSIWE) model
- creates component interfaces that can be clearly exposed, standardized for reuse, and modifiable for run-time execution
- defines the concept of a component “level” as components embedded within components
- provides a component repository for cooperative system development with strictly defined and easily accessible software component specifications [Horst 97] for efficient code reuse

From an object-oriented perspective, our system

- provides three types of objects (called “primitive components” in ControlShell): data flow components in synchronous processes, state transition components in asynchronous processes, and atomic components which can be synchronously or asynchronously executed
- defines all processing elements as objects
- constrains all objects to live within components
- automatically generates object source code with user defined execution methods and data
- provides facilities to compile user code within automatically generated object source code
- allows object inheritance and, in general, all object-oriented principles are satisfied, *e.g.*, use of object-oriented language (C++)

8 Conclusion

We have successfully demonstrated a complex inspection system that utilizes an RCS architecture and methodology supported by a component-based COTS tool called ControlShell.

The real-time lessons learned are that synchronous and asynchronous processes can operate in an RCS architecture, if the synchronous process is given higher priority. This is because the synchronous process must complete its execution each cycle. As a consequence, we gain more efficient processor usage. We also gain the ability to have more than one state transition per cycle in the finite state machines.

From the object-oriented perspective, we are fully convinced (though we have no quantitative proof) that a well-formulated architecture and methodology on top of a component-based object-oriented tool will significantly increase design, debugging, testing, and maintenance efficiency. As a qualitative measure of this claim, one engineer was able to design, debug, test, and demonstrate the CMM motion control, probe control, and vision control subsystems in about 0.5 man-years of effort, using the RCS architecture, methodology, and supporting tools. The CMM branch of the hierarchy in Figure 4 was the first branch built and tested. Later we were able to integrate and test the probe branch with relative ease and efficiency using the generic system template, the ControlShell tool set, the RCS methodology, and the architectural guidelines.

9 References

[Albus 96] Albus, J. S. and Meystel, A.M., "A Reference Architecture for Design and Implementation of Intelligent Control in Large Complex Systems," International Journal of Intelligent Control and Systems, Vol. 1, No. 1, (1996), pp. 15-30.

[Horst 97] Horst, J., Messina, E., Kramer, T., Huang, H., *"Precise definition of software component specifications"*, Proceedings of the IFAC Computer-Aided Control System Design Conference, (1997).

[Huang 96] Huang, H. and Messina, E., "NIST-RCS and Object-Oriented Methodologies of Software Engineering: A Conceptual Comparison," Proceedings of the Conference on Intelligent Systems: A Semiotic Perspective, (1996).

[Kramer 98] Kramer, Thomas R., "The NIST DMIS Interpreter: Version 2," NISTIR 6252, (1998).

[Messina 99] Messina, E., Horst, J., Kramer, T., Huang H., Tsai, T., Amatucci, E., "A Knowledge-Based Inspection Workstation," Proceedings of the 1999 IEEE International Conference on Information, Intelligence, and Systems, (1999).

[Quintero 92] Quintero, R. and Barbera, A.J. "A Real-Time Control System Methodology for Developing Intelligent Control Systems," NISTIR 4936, (1992).

[Shackleford 99] Shackleford, Will, et al, "NIST Real-Time Control Systems (RCS) Library: Tools for Control System Development," Web site address: http://eewww.eng.ohio-state.edu/nist_rcs_lib/, (1998).