Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects

David B. Stewart, *Member*, *IEEE*, Richard A. Volpe, *Member*, *IEEE*,
and Pradeep K. Khosla, *Fellow*, *IEEE*

Abstract—The port-based object is a new software abstraction for designing and implementing dynamically reconfigurable real-time software. It forms the basis of a programming model that uses domain-specific elemental units to provide specific, yet flexible, guidelines to control engineers for creating and integrating software components. We use a port-based object abstraction, based on combining the notion of an object with the port-automaton algebraic model of concurrent processes. It is supported by an implementation using domain-specific communication mechanisms and templates that have been incorporated into the Chimera Real-Time Operating System and applied to several robotic applications. This paper describes the port-based object abstraction, provides a detailed analysis of communication and synchronization based on distributed shared memory, and describes a programming paradigm based on a framework process and code templates for quickly implementing applications.

Index Terms—Dynamic reconfiguration, real-time operating system, software architecture, object-based design, port-automaton theory, reusable software, component-based design, evolutionary design, digital control systems, robotics.

1 Introduction

The port-based object (PBO) is a new software abstraction for designing dynamically reconfigurable real-time software (DRRTS). It forms the basis of a programming model that provides very specific guidelines for control engineers to create and integrate DRRTS components, yet is flexible for many types of control applications. The PBO is supported by an implementation based on domain-specific real-time operating system (RTOS) mechanisms. Together, the PBO and RTOS mechanisms form a software framework that supports the design and implementation of sensor-based control systems.

The software framework was developed as part of the *Chimera RTOS Project* [37] in the Advanced Manipulators Laboratory at Carnegie Mellon University (CMU). It is an offshoot of a project to develop reconfigurable robots [29]. We refer to the theory behind this framework as the *Chimera Methodology*. That methodology, and the corresponding RTOS mechanisms needed to support it, are the subject of this paper

The following goals for a robotics programming environment were initially set forth by several robotic projects at CMU:

- Support reconfigurable robots;
- Integrate multiple sensors;
- Enable real-time sampling rates of up to 1,000 Hz;
- Change controllers dynamically;
- Execute code transparently on multiple processors; and
- Promote collaboration in the lab through code sharing.

The Chimera Methodology is the ***solution*** to meeting the above goals. The key contributions of our solution are the following:

- A detailed definition of a port-based object, which combines the port-automaton algebraic model of concurrent processes with the software abstraction of an object, to obtain a model for creating and integrating dynamically reconfigurable real-time software components.
- Operating system services, including a PBO framework process, a multiprocessor state variable communication mechanism, and automated timing and analysis of a configuration of PBOs, that create a framework for straightforward implementation of applications that use PBOs.

In addition, perhaps the most important criterion to achieving our goals is to hide the real-time programming and analysis details. The target users of our framework are control engineers, who do not have extensive background in real-time systems or software engineering. Their strength lies in developing control systems; they want a high-level tool that is easy to use and frees them from the low level implementation details such as programming timers, analyzing schedulability, synchronizing processes, or communicating in a multiprocessor environments..

The background and the terminology used in this paper are given in Section 2. The architectural components of the framework are considered in Section 3. The focus of Section 4 is the domain-specific communication in a multiprocessor environment. Details of the PBO framework

- *D.B. Stewart is with the Department of Electrical Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. E-mail: dstewart@eng.umd.edu.*
- *R.A. Volpe is with the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109. E-mail: volpe@jpl.nasa.gov.*
- *P.K. Khosla is with the Department of Electrical and Computer Engineering and The Institute for Complex Engineered Systems, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: pkk@cmu.edu.*

Manuscript received 7 July 1993; revised 22 Nov. 1996.
Recommended for acceptance by A. Shaw.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 101183.

process are given in Section 5. Favorably, the resulting framework has many advantages that go beyond these original goals. These other advantages, which fuel our current research in this area, are summarized in Section 6.

2 Background

2.1 The Origin of the Chimera Project: Reconfigurable Robots

Robotic manipulators, such as those found on the production line of car assembly plants, have always been created with a fixed posture. However, depending on the task to be performed, some postures are unsuitable. For example, a manipulator is like a person's arm. Consider the difficulty in reaching under your car with your hand. It would be much easier to reach if your arm and hand were connected to the end of your foot. The same type of problem occurs when using robots. Different robots are better able to perform different tasks. But using multiple robots is expensive. To solve this problem, CMU developed the Reconfigurable Modular Manipulator System (RMMS) [29]. In this system, modular joints and links were created, such that they can be assembled quickly (i.e., within minutes), to create robots with various configurations. The genesis of the Chimera Project was to create the software environment for the RMMS [28].

We also identified other times when software needed to be reconfigured alongside the hardware. This included adding and removing sensors, changing the computing hardware from single processor to multiprocessing environments, and reusing generic software components with nonreconfigurable robots.

The need for dynamic reconfiguration comes from the need to change control algorithms on-the-fly, to support more intelligent control strategies. However, in reviewing the capabilities of our framework, other major advantages of dynamically reconfigurable systems emerged, including having the ability to implement real-time systems using time-based decomposition, and the ability to perform maintenance on-the-fly, especially for real-time systems that cannot be shutdown, or are too expensive to reboot. Additional benefits of our framework which go beyond the initial goals are discussed in Section 6.

2.2 Modular Components vs. Reconfigurable Components

Modular software is characterized by many guidelines, such as a simple structure, data encapsulation, functional and informational cohesion, separation of the interface specification, and the internal behavior [21], [27], [33]. The degree of modularity refers to a subjective measurement used to describe the extent that a software module follows these guidelines. For example, a system decomposed into modules may be classified as "somewhat modular" or "highly modular," depending on a software engineer's assessment of how well the module meets the defined criteria [6].

Reconfigurable components are modular components with the highest degree of modularity. Most important, it is a module designed to have replacement independence. In a modular system, there is often only one way to piece all the components together, because the interfaces of modules that need to be integrated are designed according to the other modules they interact with. For example, if you write a C or C++ module, and *#include* a *.h* file of another module, then your module becomes dependent on the interfaces of that other module. In contrast, interface specifications for reconfigurable components are designed according to a predefined standard, not according to the interfaces of other modules with which it will be integrated. Interactions between components are through these standard interfaces only.

As an example, Purtilo created the Polylith Software Bus for designing reconfigurable distributed systems [22]. Software components were made to interface with the bus, and not with other modules. The bus was implemented as a message passing transmission layer. However, the unbounded execution and blocking times of the transmission layer prevents the approach from being used directly for real-time systems. Nevertheless, Polylith demonstrates a software method for achieving replacement independence.

2.3 Static Configurability vs. Dynamic Reconfigurability

An important distinguishing feature between our approach and many other efforts in configurable systems is that of *static configurability* versus *dynamic reconfigurability* of software components.

In statically configurable systems, reusable software modules are selected and integrated off-line, and only executed after configuration is complete. Examples of these methods include real-time object-oriented programming [5], [30], software synthesis [1], [26], [31], also known as automatic code generation, interface adaptation [4], [13], [18], [23], and the Polylith Software Bus mentioned above. The static nature of these systems is a result of the need to create or generate "glue" code to integrate the components for each different configuration, then compile and link the application with this new code.

In contrast, dynamically reconfigurable systems can be modified on-line, without the need to recompile and relink the application nor shutdown and reboot the system. For example, the *Regis* environment [17] uses detailed software models and operating system services to obtain dynamic reconfigurability. Like Polylith, modules are defined according to a standard interface, rather than to other modules. Regis then uses the *Darwin configuration language*, based on the *Conic* [19] interface adaptation method, to interactively structure the components using input and output communication objects. Regis, however, has not been applied to real-time system design. One of its primary limiting factors for use in a real-time environment is that like the Polylith transmission layer, communication objects are based on message passing, with no considerations for execution or blocking times of processes. *STER* [2] is another method based on Conic, which can be used to create reconfigurable real-time systems. However, STER sacrifices the ability to perform dynamic reconfiguration in favor of providing real-time guarantees.

The research we present in this paper uses a similar concept as Regis and Polylith, by designing components according to a predefined standard interface, rather than to

the interface of other modules. Like Regis, detailed software models are defined, and operating system services are created that directly support those models. Our work, however, concentrates on many of the real-time system issues not addressed by the Polylith and Regis environments. The major differences in our work include a software abstraction that is specific to control processes, predictable communication and synchronization based on distributed shared memory, a detailed software structure that enables the automated timing of real-time properties of an application, and a programming interface that is designed especially for control engineers. In Section 5.2, we also show that static configurability is a subset of dynamic reconfigurability, and thus our solution provides the same benefits as those that only support static configurability.

2.4 Reconfigurable Software vs. Generic Software

Reconfigurable software does not necessarily imply generic software, as it is possible to have both hardware dependent and application dependent components that are not generic, but are reconfigurable. In this section, we define our classifications of reconfigurable software components. Examples of components are given later in Section 3.3.

A *generic component* is a module that is neither hardware dependent nor application dependent. The component can be configured for different types of hardware, and can be used in different applications.

Hardware dependent (HD) components are software modules that can only be executed when specific hardware is part of the system. Hardware dependent components can be of two types:

Hardware dependent interface components are used to convert hardware dependent signals into hardware independent data, such that other generic components can interface with these modules. The HD interface component is an interface to the application hardware such as robotic actuators, switches, sensors, and displays. They differ from RTOS I/O device drivers, because as processes with their own thread of control, they have the same standard interface as other software components, rather than being defined as system calls which are called by other processes.

Hardware dependent computation components provide similar functionality as generic components, but with better performance or added functionality, due to hardware-specific optimizations or modifications of the generic component. Unlike the interface components, they do not communicate directly to hardware; they are simply dependent on having specific hardware as part of the system.

Application dependent components are modules used to implement the specific details of an application. As the name implies, these components are not reusable across different applications. Ideally, these components are eliminated, since they must be redeveloped for each new application. Modules initially defined as application components, however, can often be transformed into generic components by converting hard-coded information into variable input. The input can then be obtained from the user through a teleoperating device or keyboard, from a configuration file, or from an external subsystem.

3 Architectural View of the Port-Based Object

There are several approaches to attacking the problem of creating a general programming environment for sensor-based control that meets all the goals listed in Section 1. One approach is to create the most general architecture in which every possible situation that may arise is considered. Generalizing every possible scenario is impractical. A refined version of this approach is to create a domain specific software architecture that can handle every possible situation that may arise within the domain. NASREM [3] is an example of such an architecture for the robotics domain. Our first attempt at creating a real-time processing environment was based on the NASREM model [40]. However, from our experience, using the NASREM model proved to be very difficult, partially due to the complexity resulting from trying to accommodate every possible scenario into a rigid hierarchical structure, and partially because there is no easy way to map from the theoretical architecture to a practical implementation.

We use an alternate approach, based on domain-specific elemental units. A framework is designed that uses these elemental units as building blocks to incrementally create larger, more complex applications. Various domain specific software architectures can then be created using the framework, depending on how these building blocks are ultimately assembled.

We select the *independent process* as our elemental process model.[1] An independent process does not have to communicate or synchronize with any other component in the system, and thus integration is simple. A system that is comprised only of independent components, however, is very limiting, as there are no means to share data or resources. Nevertheless, this extreme emphasizes a desire to keep the framework simple. Rather than trying to achieve the most general model of a task, we attempt to get as close to this "ideal" simple case of an independent process. The simpler each component, the simpler it will be to integrate them.

Streenstrup, Arbib, and Manes formalized the algebra of independent concurrent processes with their port-automaton theory [32]. They model a concurrent process as an independent automaton, which operates on the state of the environment.

When a process needs information, it obtains the most recent data available from its ***input ports***. This port can be viewed metaphorically as a window in your house; whatever you see out the window is what you get. There is no synchronization with other processes and there is no knowledge as to the origin of the information that is obtained from this port.

When a process generates new information that might be needed by other processes, it sends this information to its ***output ports***. An output port is like a door in your home; you can open it, place items outside for others to see, then close it again. As with the input ports, there is no synchronization with other processes, and there is no knowledge as to which processes might look at this information.

1. Although the term *process* is used throughout this paper, implementation in our RTOS is done using *lightweight processes*, which are also called *threads* in many operating systems.

Lyons and Arbib [16] applied this model specifically to robotics, and showed that a stable control system can be achieved using a formally specified language they termed Robot Schemas. As compared to the Robot Schemas model, we extend the port-automaton model to include multiple types of ports, and we create a framework for directly implementing software components using this computational model.

In addition to the independent process, we select the *object* as an elemental software abstraction. As stated by Wegner, an object is the atomic unit of encapsulation, with operations that control access to the data [42]. The term *object* does not imply "object-oriented design," which is an extension to objects to include *classes* and *inheritance*. The references to objects in this proposal are thus classified as *object-based design*, as defined by Wegner's distinction of that term and *object-oriented design* [43].

3.1 The Port-Based Object

We combine the algebraic model of a port automaton with the software abstraction of an object, to create the ***port-based object (PBO)***, shown in Fig. 1. In our diagrams, we draw a PBO as a round-corner rectangle, with input and output ports drawn as arrows entering and leaving the side of the rectangle. Configuration constants are drawn as arrows entering/leaving the top of the rectangle. Resource ports are shown as arrows entering/leaving the PBO from the bottom.

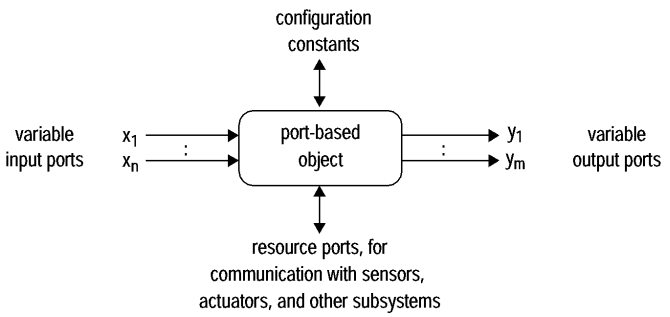


Fig. 1. Architectural view of a port-based object.

A PBO is an independent concurrent process, whose functionality is defined by the methods of a standard object. Communication with other modules is restricted to its *input ports* and *output ports*, as defined by the port-automaton theory. There is no explicit synchronization with other processes. The *configuration constants* are used to reconfigure generic components for use with specific hardware or applications.

In addition to input and output ports, we also define *resource ports*, which are needed to create an environment for multisensor integration. The resource ports connect to sensors and actuators, via I/O device drivers, which are not PBOs. The details of accessing the sensor or actuator are thus encapsulated within the PBO, resulting in an HD interface component (as defined in Section 2.4).

By modeling PBOs to have optional configuration constants and resource ports, we have been able to use the same PBO model for all types of reconfigurable components. A sample library of PBO objects for robotic manipulators is shown in Table 1. The library represents a subset of PBOs that were created in our laboratory at CMU.

An important note about the functional descriptions of the modules is that *the framework is designed independent of the granularity of functionality in each PBO*. The software architect who decomposes an application into modules defines the granularity; our framework then provides the mechanisms for quickly realizing each of these modules by using the PBO model to implement them as reconfigurable objects.

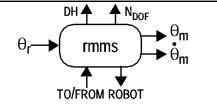
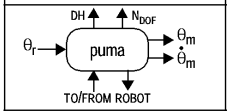
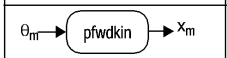
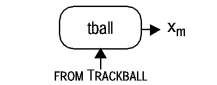
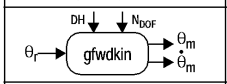
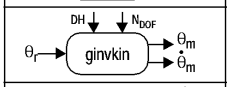
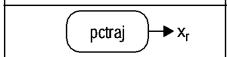
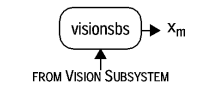
Similarly, *the framework does not define the type or semantics of the port variables*. A variable type mechanism [38] is used so that data transmitted over the ports can be any type. For example, it can be raw data, such as input from an A/D or D/A converter; processed data, such as positions and velocities; or processed information, such as structures describing types and locations of objects in the environment.

In our implementation, a configuration file is used to specify the information shown in the *PBO Name and Ports* column of Table 1. For example, the configuration file *puma.rmod* for module *puma* is shown in Fig. 2. The MODULE line specifies the name of the object module file name (extension omitted) of the code used for this PBO. Multiple PBOs can specify the same code, for cases where multiple versions of the same object, possibly each with its own unique configuration, are desired. The DESC line gives a one-line description of the module. The INVAR and OUTVAR lines define the input and output ports, respectively, the INCONST and OUTCONST lines define the configuration constants. Since ASCII characters for the configuration file restrict us, we created our own conventions for mapping names, as shown in Table 2. Note that these conventions are personal preferences, and not specifications of the framework. TASKTYPE is either *periodic* or *aperiodic*. The FREQ line is the initial frequency at which a periodic process corresponding to this PBO executes; the frequency can be changed dynamically. The SVARALIAS line is used for mapping internal and external names, as described in the next section. The LOCAL line marks the beginning of module-specific information. Any information after this line is not used by the PBO framework, but rather passed on to the initialization code of the PBO. Lines beginning with # are comments and are ignored.

3.2 Configurations

As defined by Dorf [7], "a control system is an interconnection of components forming a system configuration that will provide a desired system response." Each component can be mathematically modeled using a transfer function, which computes an output response for any given input response. The port-automaton theory provides an algebraic model for these types of control systems. By incorporating the model into the PBO, our PBOs also provide this same model suitable for control engineers. PBOs are configured to form a control system in the same way as a control engineer configures a system using transfer functions and block diagrams. This approach allows us to satisfy an important criterion for our DRRTS framework, to make a framework for control engineers, rather than for software or real-time system engineers.

TABLE 1
EXAMPLE OF SOFTWARE COMPONENTS IN A ROBOTIC MANIPULATOR PBO LIBRARY

PBO Name and Ports	Type	Description
	HD interface	Interface to the Reconfigurable Modular Manipulator System (RMMS)
	HD interface	Interface to Puma 560 robot, using the Trident Robotics TRC004 [12] hardware to bypass the VAL controller
	HD computation	Compute Forward Kinematics for a Puma 560 Robot.
	HD computation	Compute Inverse Kinematics for a Puma 560 Robot.
	HD interface	Interface to a 6-degree-of-freedom trackball.
	generic	Compute generalized Forward Kinematics based on the DH-parameters obtained during initialization of the module.
	generic	Compute generalized Inverse Kinematics based on the DH-parameters obtained during initialization of the module.
	generic	Given the current measured position and the desired final position, compute new intermediate reference positions and velocities for each cycle.
	application	Move manipulator according to the previously computed application-specific trajectory.
	application	Interface to external vision subsystem that supplies next reference position based on visual tracking

```
MODULE          puma
DESC            Puma 560 controller
SVARALIAS       Q_REF=Q_IN
INCONST         none
OUTCONST        NDOF DH
INVAR           Q_REF
OUTVAR          Q_MEZ Q^_MEZ
TASKTYPE        periodic
FREQ            500

LOCAL
KPGAINS         4000.0  11000.0  3000.0  500.0  310.0  300.0
KVGAINS         80.0    114.0    25.0    25.0   12.0   17.0
DEVICEFILE      pumadC

#  maximum joint error in radians; robot shuts down if exceeded
#  this field is optional; default value is 0.05 (conservative)
MAX_JOINT_ERR   0.05 0.05 0.05 0.05 0.05 0.05
EOF
```

Fig. 2. File *puma.rmod*: sample configuration file for *puma* PBO.

TABLE 2
SUMMARY OF INVARS, OUTVARS, AND CONFIGURATION CONSTANTS FOR SAMPLE LIBRARY OF TABLE 1

Variable	ACII Name	Description	Type	Size
θ_r	Q_REF	Reference Joint Position	floats [N_{DOF}]	4 x N_{DOF}
θ_m	Q_MEZ	Measured Joint Position	floats [N_{DOF}]	4 x N_{DOF}
$\dot{\theta}_m$	Q^_MEZ	Measured Joint Velocity	floats [N_{DOF}]	4 x N_{DOF}
x_r	X_REF	Reference Cartesian Position	floats [6]	24
$\dot{x}_r$	X^_REF	Reference Cartesian Acceleration	floats [6]	24
x_m	X_MEZ	Measured Cartesian Position	floats [6]	24
x_d	X_DES	Desired Cartesian Position	floats [6]	24
DH	DH	Denavit Hartenberg Parameters	float [4][N_{DOF}]	16 x N_{DOF}
N_{DOF}	N_DOF	Number of Degrees of Freedom	short int	2

A *configuration* is a set of PBOs which are interconnected to provide the required open- or closed-loop system. In our implementation, the set of PBOs can be specified in one of four ways using a graphical software assembly tool, using a command-line interface, through a network from an external planning subsystem, or embedded using the C programming language.

A configuration is valid only if for every PBO selected, any data that it requires at its input ports is produced by one of the other PBOs as output. As per the port-automaton theory, the control engineer does not have to be concerned with how data gets from the output of one PBO to the input of another PBO. The communication is embedded in the framework, such that it is transparent to the control engineer. A configuration also cannot have two PBOs that produce the same output; otherwise there may be a conflict as to which output should be used at a given time.

Port names are used to perform the bindings between input and output ports. Whenever two PBOs exist with matching input and output ports, the framework creates a communications link from the output to the input. If necessary, the output can be fanned into multiple inputs. Our framework uses an internal/external name separation for the ports, such that the name used to code the PBO can be independent of the name used for linking that object to other PBOs. The mapping between internal and external name is done in the SVARALIAS lines of the configuration file for each PBO. The left value represents the name used in the configuration, and the right value is the name used internally by the PBO. If an SVARALIAS line is not specified, then the default is that both the internal and external names are the same.

The correctness of a configuration is verified analytically using set equations, where the elements of the sets are the state variables. If X_j is a set representing the input variables of module j, Y_j is a set representing the output variables of module j, then a configuration is legal only if the following two conditions are true:

$$\left(\left(\bigcup_{j=1}^{k} X_j\right) \subseteq \left(\bigcup_{j=1}^{k} Y_j\right)\right) \quad (1)$$

$$\left(Y_i \cap Y_j\right) = \emptyset, \text{ for all } i, j \text{ such that } 1 \le i, j \le k \wedge i \ne j \quad (2)$$

where k is the number of modules in the configuration. Equation (1) represents our first condition that there must be a corresponding output port for every input port. Equation (2) ensures that two PBOs do not produce the same output.

3.3 Configuration Examples

In this section, we illustrate through examples how to create configurations out of PBOs stored in a library. Details of designing individual PBOs are given later in Section 5. In our implementation, a configuration can be assembled graphically using the Onika visual programming environment [8].

3.3.1 Cartesian Control of the Reconfigurable Modular Manipulator System

Fig. 3a shows a configuration, using modules from our sample library shown in Table 1, to perform teleoperated Cartesian control of the RMMS. The configuration of the RMMS robot is not known beforehand. Rather, its configuration is read from EPROMs embedded in the robot during initialization. From that configuration, the *rmms* module outputs the N_{DOF} and *DH* configuration constants. Those constants are used as input to the *gfwdkin* and *ginvkin* modules, which can be configured for any robot based on N_{DOF} and *DH* [11]. A teleoperation interface is provided by the 6-DOF trackball, and the *cinterp* module is used to generate intermediate trajectory points for the robot, because the *tball* module typically executes at a much lower frequency than the other modules.

The software framework does not pose any constraints on the frequency of each PBO. Rather, as defined by the port-automaton theory, every PBO is an independent concurrent process that can execute at any frequency. Whenever that process needs data from its input ports, it retrieves the most recent data available. When it completes its processing, it then places any new data onto its output ports.

A configuration can be executed in either a single- or multiprocessor environment. In a multiprocessor environment, the control engineer only needs to specify which processor to use for each PBO. The communication between PBOs and synchronization of their processes is otherwise identical, and fully transparent to the control system engineer, as detailed in Section 4.

3.3.2 Cartesian Teleoperation of a Puma 560

Suppose that a Puma 560 robot is to be used instead of the RMMS. The *rmms* module can be replaced with the *puma* robot interface module, as shown in Fig. 3b. Since the Puma is a fixed configuration robot, its N_{DOF} and *DH* parameters are constant. Instead of reading these values from the robot, they can instead be hard-coded into the *puma* module, and output as configuration constants. There is no need to change any other module, since the *gfwdkin* and *ginvkin* modules will configure themselves during initialization for the Puma based on the new values of N_{DOF} and *DH*.

3.3.3 Improving Performance of a Puma 560

Generic components are useful for enabling rapid prototyping, but they may not always be computationally efficient. For example, the computation of the forward kinematics based on the *DH* configuration constants and using matrix operations will naturally be slower than performing similar computations for a specific robot, such as the Puma 560. For a fixed-configuration robot, the *DH* parameters are constant, and unnecessary computations (such as multiply by zero or 1, or computing $sin(\pi/2)$) can be eliminated.

To improve the performance of an application, an HD computation components can be created. The *pfwdkin* and *pinvkin* modules are examples of such components. They compute the forward and inverse kinematics specifically for a Puma 560, and they execute faster than their generic counterparts. It is then desirable to replace *gfwdkin* with *pfwdkin*, and *ginvkin* with *pinvkin*, as shown in Fig. 3c, whenever the *puma* HD interface component is used.

In order for an HD computation component to replace a generic component, it must provide at least the same outputs and must not require any additional inputs as compared to the generic component. Even when an HD component is

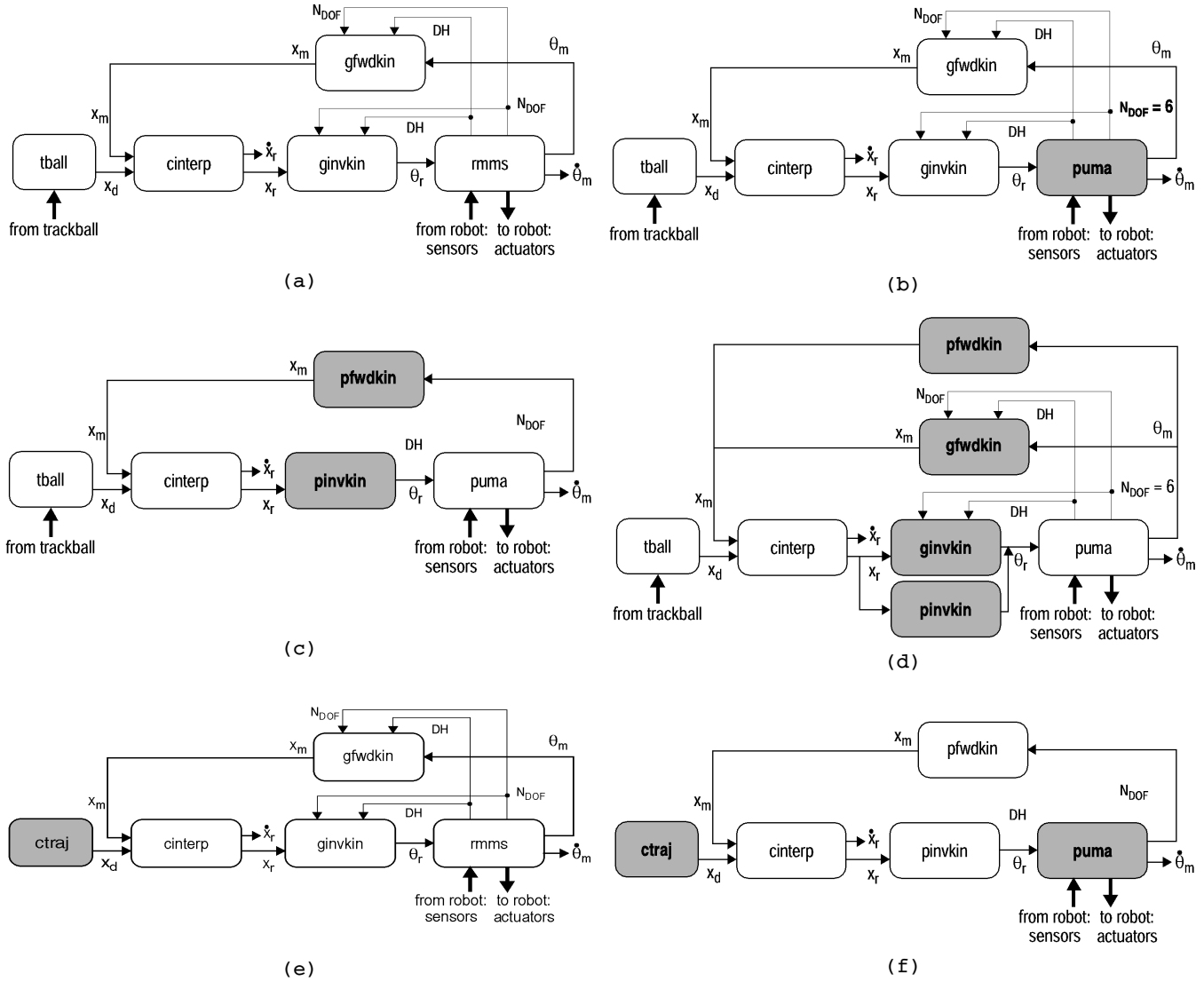


Fig. 3. Example of component-based design using port-based objects. (a) example of Cartesian teleoperation of the RMMS using GCs; (b) example of Cartesian teleoperation of a Puma 560 using GCs; (c) example of Cartesian teleoperation of a Puma 560 using HDCSs; (d) example of fault-tolerant components, with co-existence of old and new software components; (e) example of application-specific autonomous execution of the RMMS using GCs; (f) example of application-specific autonomous execution of a Puma 560 using HDCCs.

used, it does not eliminate the usefulness of the generic component. For example, in order to improve fault tolerance of an application, the generic components can still be used as standby modules, or as shown in Fig. 3d, it can execute in parallel with the HD computation components, albeit at lower frequencies, in order to provide consistency checks.

3.3.4 Autonomous Execution of a Puma 560

As an example of an application component, suppose that a custom autonomous trajectory module *ctraj* is created to replace the teleoperation module *tball*, as shown in Fig. 3e. The component can be integrated into the system by defining it as a port-based object.

Even though a module is application dependent, it does not have to be hardware dependent. Thus, if the hardware for the application is changed, the application component does not necessarily have to change. Fig. 3f shows this by replacing *rmms* with *puma*, but not changing the trajectory of the robot's end effector, as defined by *ctraj*.

3.3.5 Cartesian Control of a Torque-Mode Robot

As a more elaborate example of a configuration, a telerobotic Cartesian visual servoing subsystem is shown in Fig. 4 (note that these modules are from a different PBO library than the one defined in Table 1). Input can come from either a user through a trackball or from an external vision subsystem. The port interfaces of each PBO can easily be determined simply by looking at the inputs and outputs of each object. Despite the seemingly complex communication paths between objects, communication remains transparent from the control system engineer, and lines are simply drawn between matching input and output ports.

The configuration of PBOs is not the only part of a subsystem. As shown in Fig. 5, PBOs can interface with device drivers, external subsystems, software libraries, special purpose processors, and user interfaces. In this paper, however, we focus on the configurations of port-based objects, and the associated communication, synchronization, and analysis.

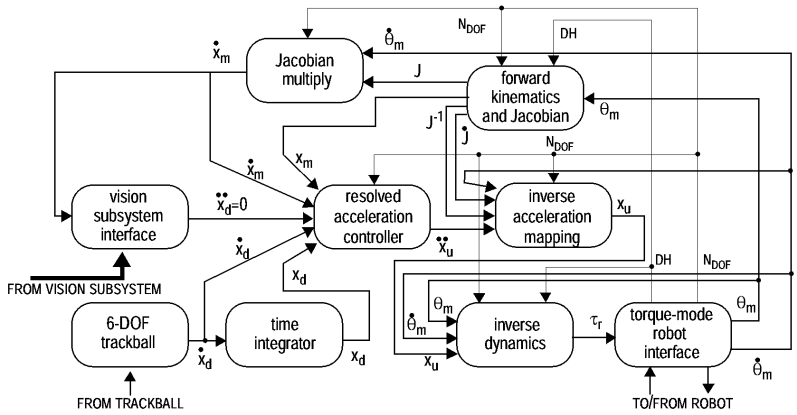


Fig. 4. Example of module integration: Cartesian teleoperation.

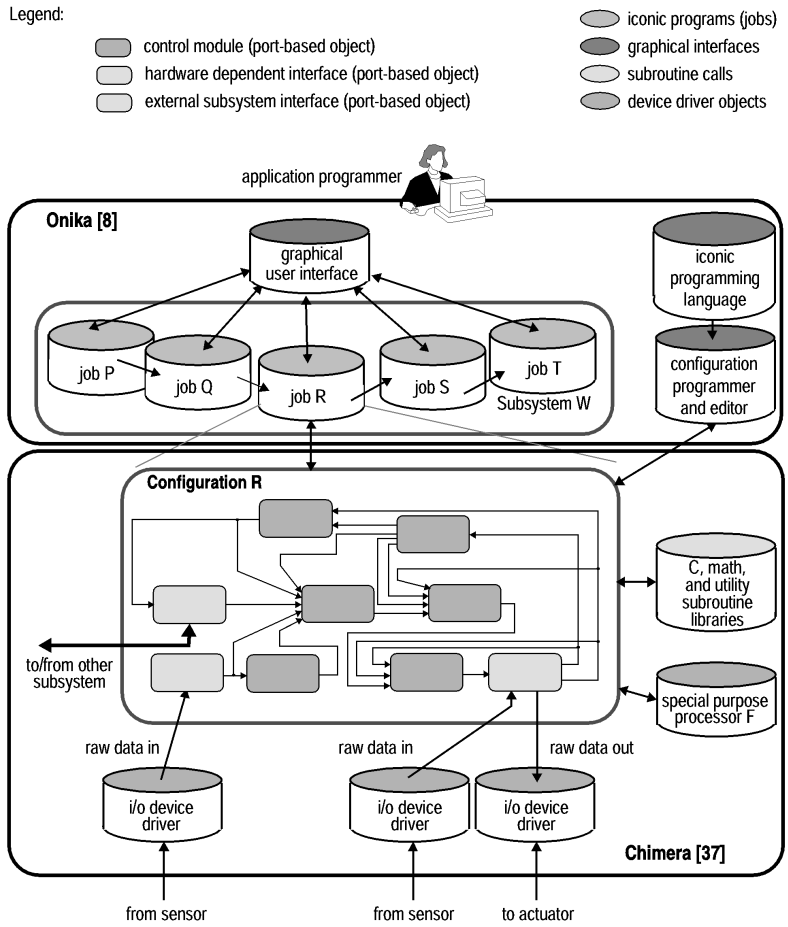


Fig. 5. Complete software infrastructure for a sensor-based control subsystem.

4 Interobject Communication

Integrating software components such that all communication is performed in a predictable and timely manner is perhaps the most difficult aspect of creating reconfigurable real-time systems. In order to support the port-based object model, we need a communication mechanism that meets the following requirements:

- Support the port-automaton model of independent processes. That is, read the input ports at the beginning of each cycle to obtain the most recent data available, and write to the output ports at the end of each cycle.
- Disallow synchronization or communication between processes except through ports.
- Target data transfers with low volume but at high frequency (1,000 Hz).
- Have a simple and straightforward binding scheme for communication links to dynamically reconfigure a subsystem in bounded time.

- Fan an output into multiple inputs.
- Support transparent multiprocessing, so that the processes that are communicating can be either on the same or different processors, without any difference in the communication.
- Allow communication between processes that may be executing at different frequencies.

We designed a communication mechanism that meets all of these requirements, based on the combined use of local and distributed shared memory. The domain-specific solution takes advantage of several of the characteristics of control systems, such as the assumption on transfer rates and the need to only read the most recent data, rather than all data. The solution provides a level of performance and predictability that has not previously been achieved using more general message-passing mechanism.

Port communication has often (and more typically) been implemented using some form of message passing. This alternative was considered, but not selected for several reasons:

- Using messages, we could not support the port-automaton theory because the most recent data is not always readily available. For example, if the process producing the data is faster, then the messages may be queued, and the message received by the consumer might not contain the most recent data.
- Fanning an output to multiple inputs is difficult because it requires a message to be duplicated for each input or requires a more complex mechanism to ensure that messages are not deleted until all processes needing it have used it. Duplicating messages based on the number of recipients also violates the port-automaton theory, which states that a process is unaware of the destination of the data on its output ports.
- The overhead with sending messages, especially in a multiprocessor environment, is much higher than that achievable using shared memory. This factor is especially important considering some data must be transferred 1,000 times per second.

These drawbacks of message passing systems led to our design of a mechanism based on distributed shared memory. Our work focuses on loosely coupled shared memory architectures.

4.1 State Variable Communication

The communication between PBOs is performed via state variables stored in global and local tables, as shown in Fig. 6. Every I/O port and configuration constant is defined as a state variable (SVAR) in the global table, which is stored in shared memory.

A PBO can only access the local table, where only the subset of data from the global table that is needed by that PBO is kept. Since every PBO has its own local table, no synchronization is needed to read from or write to it. A PBO process can, thus, execute independently of other processes by using the data in its local table. Consistency between the global and local tables is maintained by the SVAR mechanism, as detailed in the remainder of this section. As an example, Fig. 7 shows the contents of the global and local tables for the sample configuration that was illustrated in Fig. 3a.

Support for SVAR communication has been built into our framework, such that updates of the local and global tables occur at predetermined times only. Configuration constants are updated only during initialization of the PBO. The state variables corresponding to input ports (which we call INVARS) are updated prior to executing each cycle of a periodic PBO, or before processing each event for an aperiodic PBO. During its cycle, a PBO may update the state variables corresponding to output ports (which we call OUTVARS) at any time. These values are only updated in the global table after the PBO completes its processing for that cycle or event. All transfers between the local and global tables are block transfers (i.e., using a routine like UNIX's *memcpy()*). Ensuring the integrity of the data is a matter of ensuring that the block transfers are performed as critical sections.

Although there is no explicit synchronization or communication among processes, we must ensure that accesses

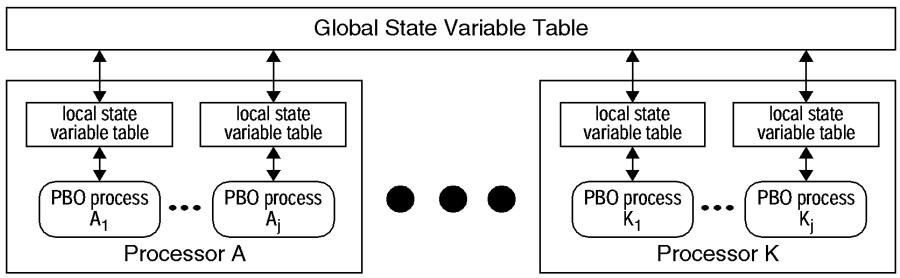


Fig. 6. Structure of state variable table mechanism for port-based object integration.

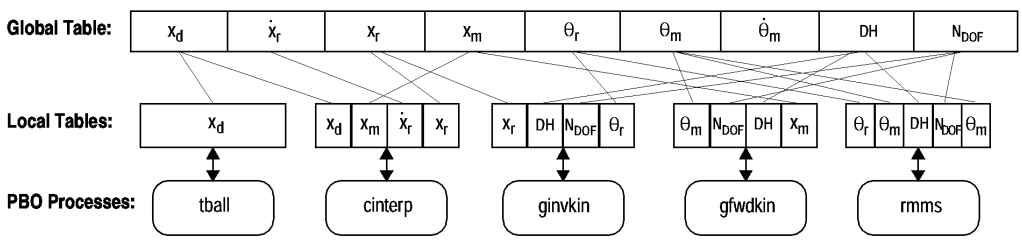


Fig. 7. Contents of global and local tables for sample configuration shown in Fig. 3a.

to the same SVAR in the global table are mutually exclusive, which creates potential implicit blocking. The locking mechanism we use enables us to maintain the autonomous execution model of the PBO, while also ensuring the integrity of the communication.

4.2 Locking the Global SVAR Table

The method we use to lock the global table and preserve the autonomous execution model of the PBO is based on the assumption that the amount of data communicated via the ports on each cycle of a PBO is relatively small. That is, each INVAR or OUTVAR is only a few tens to a few hundreds of bytes. This is in contrast to communication systems where images may require many thousands or millions of bytes per cycle. The exact value of what we mean by "small" depends on a particular configuration, and is quantified as part of our analysis in Section 4.4. For this reason, our framework, which defines objects that are specific to the control systems domain, cannot directly be applied to a general communication system. However, as was shown in Fig. 4, we can use the external subsystem interface to interact with a communication system.

Processes executing on different CPUs can access the global table, and therefore a multiprocessor solution for locking the table is required. The type of multiprocessor synchronization we require for the global SVAR table has been addressed in [24]. The *shared memory protocol* (SMP) is presented as an extension of the single processor priority ceiling protocol [25]. The protocol involves defining global semaphores for locking the global shared memory and placing priority ceilings on accessing the semaphores to bound the waiting time of higher priority jobs.

Unfortunately, there are several problems that prevent the use of SMP within our framework:

- This method assumes that the local scheduling on each processor is the rate monotonic algorithm, with static priorities. As discussed in [39],it is desirable to use mixed or dynamic priority algorithms for scheduling reconfigurable systems, for which the protocol is not suitable.
- One assumption of the SMP is that the delay to access the backplane bus from any processor is negligible compared to the task execution times. Unfortunately this is usually not the case; buses like the VMEbus are implemented using a static-priority processor assignment that is not under the control of software. Therefore, the time to wait for the bus can be significant if a process is on a low-priority CPU.
- There is significant overhead associated with implementing SMP that prevents its use with control applications requiring frequencies over 1,000 Hz. The complexity and overhead of SMP can be reduced significantly for the port-based communication by selecting a single lock for the entire table, instead of a separate lock for each state variable. Selecting a single lock for the entire table is not as restrictive as it seems, since a shared bus connects the shared memory to local memory. Even if multiple tasks have separate locks only one of them can physically access the shared memory at once; other tasks must wait for the bus even while in their critical section.

An alternate solution for synchronizing access to the global state variable table is to use spin-locks [20]. When a task must access the global table, it first locks the processor on which it is executing. Locking the CPU ensures that the task does not get swapped out while holding the critical global resource. The task then tries to obtain a global lock by performing an atomic *read-modify-write* instruction, which is supported by most hardware processors. If the lock is obtained, the task reads or writes the global table then releases the lock, still while being locked into the local CPU. It then releases its lock on the local processor. If the lock cannot be obtained because it is held by another task, then the task spins on the lock. It is guaranteed that the task holding the global lock is on a different processor, and will not be preempted, thus it will release the lock shortly.

In theory, locking the CPU can lead to possible missed deadlines or priority inversion. However, considering the practical aspects of real-time computers, it is not unusual that a real-time microkernel locks the CPU for up to 100 μsec in order to perform system calls such as handling timer interrupts, scheduling, and performing full context switches [37]. Furthermore, many RTOS are created such that periods and deadlines of processes are rounded to the nearest multiple of the system clock since more accurate timing is not available to the scheduler. In these systems, if the total time that a CPU is locked to transfer a state variable is small as compared to the resolution of the system clock, then there is negligible effect on the predictability of the system due to this mechanism locking the local CPU.

In comparing this method to SMP, the lock can be viewed as a single global semaphore, and since all tasks can access it, its priority ceiling is constant, and is equal to the maximum task priority in the system. Since there is only one lock there is no possibility of deadlock. A task busy-waits with the local processor locked until it obtains the lock and goes through its critical section. In Section 4.4 it is shown that for configurations where the volume of data transferred between objects is small, there is a bounded waiting time for obtaining the global lock, even on hardware such as the VMEbus that only has fixed priority bus arbitration.

4.3 Initialization

The global table is initialized in shared memory based on the contents of an SVAR configuration file. The file includes the names and types of all the INVARS, OUTVARS, and configuration constants. A default configuration should be provided in conjunction with each PBO library, which defines the name, type, and size of each variable. For example, Table 2 shows the information that would be included in the SVAR configuration file for the library that was shown in Table 1. For any application, a programmer may update this default configuration file by adding new variables to it if they make use of the SVARALIAS facility to change some of the port names, or by deleting variables that are not required for the application.

A local table is only created and initialized when a PBO is created, and contains those SVARs given in the *.rmod* configuration file (as was shown in Fig. 2). Information about the type and size of those SVARs is retrieved from the global table, which is why it did not have to be specified in the *.rmod* file. For each SVAR, a pointer is also stored in the local table,

which points to the corresponding SVAR in the global table, and used during the block transfer operations.

Configuration constants create a necessary order of initialization for PBOs. Any PBO that has an output configuration constant (which we call OUTCONST), must be initialized before any other PBO with the same constant as an input (called an INCONST) is created. INVARS and OUTVARS, on the other hand, do not have a necessary order, since in control systems they often form a closed loop system, and thus the order of initialization of PBOs is not obvious. Beyond the necessary order for constants, ordering initialization of PBOs in a closed loop system to ensure stability of the control system is generally application specific, and thus not discussed further within our framework. Our framework, however, can support any order of initialization of the PBOs, as dictated by the application.

4.4 Analysis of SVAR Mechanism

In this section, we show that on a fixed priority hardware platform such as the VMEbus, it is possible to provide predictable high performance communication using the SVAR mechanism. The analysis assumes that data from the input ports is transferred once from the global table to the local table at the beginning of a process' cycle, and data destined for the output ports is transferred from the local table to the global table upon completion of the process's cycle. The PBO framework ensures that communication occurs at these specified times, as described in Section 5.

4.4.1 Transfer Times

To ensure predictable communication, the time required to transfer data between the local and global tables for each task must be computed. Let t_{IP} be the time required to transfer the *INVARS* and t_{OP} be the time required to transfer the *OUTVARS* of a PBO *P*, assuming no waiting for the bus. These values are computed as:

$$t_{IP} = V_1 + n_{IP}V_a + \sum_{i=1}^{n_{IP}} R(x_{Pi})$$
$$t_{OP} = V_1 + n_{OP}V_a + \sum_{i=1}^{n_{OP}} R(y_{Pi}) \quad (3)$$

where V_1 is the overhead for locking and unlocking the table, excluding waiting time for the bus. V_a is the overhead of transferring each additional variable; n_{IP}/n_{OP} are the number of *INVARS/OUTVARS* for object *P*; x_{Pi} / x_{Po} = number of transfers required for the *INVAR/OUTVAR i* of object *P*; and *R(x)* = time required for *x* transfers. V_1, V_a, and *R(x)* are dependent on the speed of the hardware. These values can be measured initially for each type of hardware supported, then used by a configuration manager for estimating communication times. As an example, V_1, V_a, and *R(x)* were measured in our laboratory. The breakdown of times for an Ironics IV3230 single board computer [9] with a 25 MHz MC68030 processor on a VMEbus is shown in Table 3. A VMETRO 25 MHz VBT-321 VMEbus analyzer [41] was used to time the communication, and provided a resolution of better than 1 μsec. The global state variable table was stored within the dual-ported memory of a second IV3230.

Note that the value of *R(x)* is not linear. This is due to the underlying block copy routine, which has a better average time per transfer for larger transfers. Through interpolation, different transfer sizes can be estimated, and more measurements of *R(x)* with different values of *x* can give more accurate results. However, for purposes of discussion and examples in this paper, the values shown are sufficient.

The values in Table 3 can be substituted into (3) to estimate the transfer times for each port-based object. As an example, consider the joint control of a robot with built-in controllers, whose configuration is shown in Fig. 8, and assume that N_{DOF} = 6. The values of t_{IP} and t_{OP} for each module were estimated. These estimates were then compared to actual transfer times measured with the VMETRO analyzer. As can be seen in Table 4, the estimates and actual times are sufficiently close to use the estimates for further analysis. This aspect is important since it is not desirable, and perhaps not feasible, to measure the communication of every software module for every type of hardware. For simplicity, (3) assumes that V_1 is always present, even if *n* is 0, such as n_{IP} for *jtball*. In practice, if there are no transfers to be made, the global table is not locked. As a result, the actual measured time is very small and accounts for overhead of testing if a transfer must be made.

4.4.2 Waiting Time for Global Table Lock

Until now, we assumed there is no contention for the global table's lock. Next, we compute the worst-case waiting time for the lock by each task. Let L_{pj} be the maximum time that task *p* on processor *j* will hold the global table lock. Therefore L_{pj} = max(t_{Ip}, t_{Op}). Let M_j be the longest time that the global lock is held by any task on processor *j*, then

$$M_j = \max\left(L_{pj}\Big|_{p-1}^{N_j}\right) \quad (4)$$

where N_j is the number of tasks on processor *j*.

Ideally, if multiple tasks are trying to obtain the lock, the one with the highest priority succeeds. Unfortunately, on a shared bus where each processor has a fixed priority, such as the VMEbus that is not using round-robin bus arbitration, that is not the case. Instead, the task inherits the priority of the processor. For the remainder of the analysis, assume that the hardware is a fixed-priority VMEbus, such that the lowest numbered processor has highest priority. For different hardware configurations, the following analysis may have to be redone, and perhaps a different form of locking for the global table may be appropriate.

Any task on processor *k* attempting to lock the global table must wait for tasks on all higher priority processors. Furthermore, the task may also have to wait for a task currently holding the lock on a lower priority processor. Based on the locking mechanism described in Section 4.2, only one task on any processor can request the lock at once, and therefore there is no contention with other tasks on the same processor.

In Section 4.2, an assumption was made that the state variable table mechanism was valid as long as the amount of data to be transferred is small. That assumption is now quantified, as the volume of data affects the maximum waiting time of each task. Let W_k be the worst case waiting

TABLE 3
BREAKDOWN OF VMEBUS TRANSFER TIMES AND COMMUNICATION OVERHEAD

Operation	Execution Time (μsec)	Variable in equation (3)
obtaining global state variable table lock using TAS	5	
releasing global state variable table lock	2	
locking CPU	8	
releasing CPU lock	8	
initial subroutine call overhead	4	
block copy subroutine call overhead	7	
total overhead for single variable read/write	**34**	V_1
additional overhead, per variable, for multivariable copy	5	Va
raw data transfer over VMEbus, 6 oats	9	$R(6)$
raw data transfer over VMEbus, 32 oats	31	$R(32)$
raw data transfer over VMEbus, 256 oats	237	$R(256)$

TABLE 4
COMPARISON OF ESTIMATED AND ACTUAL TRANSFER TIMES

module	n_{IP}	n_{OP}	x_{Pi}	y_{Pi}	Estimated t_{IP}	Actual t_{IP}	Estimated t_{OP}	Actual t_{OP}
puma_pidg	3	2	6	6	76	67	64	54
grav_comp	1	1	6	6	41	40	41	40
diff	1	1	6	6	41	40	41	40
jtball	0	1	0	6	34	2	41	40

TABLE 5
SAMPLE COMPUTATIONS OF WORST EXECUTION TIME (ALL TIMES IN msec)

Module	Processor	Task ID	Frequency (1/T), in Hz	Period (T)	WCET (C), Excluding Waiting	W_{kLO}	W_{kHI}	W_k	Adjusted WCET ($C+W_k$)
puma_pidg	1	τ_1	1000	1.0	0.25	0.041	0	0.041	0.29
grav_comp	1	τ_2	300	3.3	1.20	0.041	0	0.041	1.24
diff	2	τ_1	500	2.0	0.80	0	0.222	0.222	1.02
jtball	2	τ_2	20	50.0	20.0	0	0.222	0.222	20.22

time for any task on processor k. Since this is *waiting* time and not *blocking* time (a *waiting* task is in the running state, a *blocked* task is suspended) W_k can be added to the worst-case execution time of a task. It is computed as

$$W_k = W_{kLO} + W_{kHI} \tag{5}$$

where W_{kLO} and W_{kHI} are the maximum time the task may have to wait for a task to release the lock on a lower or higher priority processor, respectively. W_{kLO} is computed simply as the longest time any single task on a lower priority processor may hold the lock. Therefore,

$$W_{kLO} = \max\left(M_j \Big|_{j=k+1}^{r}\right) \tag{6}$$

where r is the number of processors.

Next, W_{kHI} is computed. For $k = 1$, there are no higher priority processors, thus $W_{1HI} = 0$ and $W_1 = W_{1LO}$. For $k > 1$, the potential locking of the table for all tasks on processors 1 to $k - 1$ must be considered. Under the assumption that the volume of data is small, the bandwidth required to transfer all the data is much less than the total bandwidth of the bus. Therefore, in the worst case, all tasks on higher-priority processors may require the lock at the same time. W_{kHI} is thus computed as the sum of the waiting time of all tasks on higher-priority processors as:

$$W_{kHI} = \sum_{j=1}^{k-1} \sum_{i=1}^{N_j} \left(t_{I,ij} + t_{O,ij}\right). \tag{7}$$

The notation $t_{I,ij}$ is the same as t_{IP}, where a process P is referred to by the processor number i and task ID j. As an example, (5), (6), and (7) were applied to the sample configuration that was shown in Fig. 8, with estimated locking times as were shown in Table 4. Task periods and cycle times (before adding maximum waiting time) were arbitrarily assigned to illustrate the computations. Assuming that *puma_pidg* and *grav_comp* are on processor 1, and *diff* and *jtball* are on processor 2, the resulting computations are shown in Table 5. The adjusted execution time should be used in a schedulability analysis.

In our applications, the average case is significantly lower than the worst case. To compensate, a soft real-time scheduling algorithm can be used to schedule the less critical tasks on lower priority processors [39], while hard real-time critical tasks should be placed on the high-priority processors. Another consideration for assigning tasks to processors is the volume of data that needs to be transferred. The computations of W_k show that it is preferable for tasks producing a low volume of data to be placed on higher priority processors, since that significantly reduces

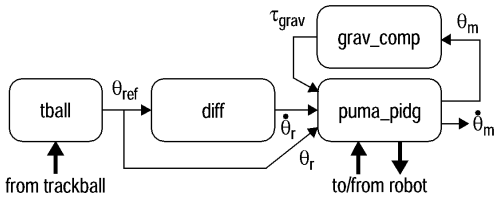


Fig. 8. Joint control of a robot manipulator with built-in PID controller.

W_{kHI} for tasks on lower priority processors. A different assignment of tasks to processors can lead to very different results. A configuration manager can perform the above analysis for various configuration possibilities, in order to optimize the global allocation of processes to processors.

5 Framework Process for Implementing Port-Based Objects

The analysis in previous sections was performed with an underlying assumption that the framework handles communication and synchronization. In this section, we look at the details of the PBO, and show how we have implemented the framework as part of the RTOS. This allows a control engineer to easily implement individual software components using the PBO model.

5.1 An Inside-Out Programming Paradigm

Creating the code for PBOs is an "inside-out" programming paradigm as compared to traditional coding of real-time processes, as shown in Fig. 9. The grey area shows RTOS code, while the black areas show the user code. The traditional approach is used by most current RTOS; processes are created, each with their own *main()* (or equivalent function name). The process executes user code and controls the flow of the program. It invokes the operating system, typically via a system call, whenever an OS service is required. OS services include communication, synchronization, programming timers, and creating new processes.

The PBO method, on the other hand, provides a consistent structure for every process, and thus operating system services such as communication, synchronization, scheduling, and process management are performed in a predictable manner. Only when necessary, the operating system calls a PBO's method to perform user-defined functions. This predefined structure also allows the RTOS to continually measure the execution time of the PBO code, using an automated task profiling mechanism as described in [35].

A PBO process is realized by creating a single, standard process, which we call the framework process (*pboframe()*). Every process in the system uses this same framework, and takes a PBO as an argument. The PBO defines the module-specific user code, the I/O ports and configuration constants, the type of process (e.g., *periodic process* or *aperiodic server*), and the timing parameters such as frequency, deadline, and priority.

5.2 The Framework Process

The framework process implements a finite state machine with four states, as shown in Fig. 10. The states are shown as bold ellipses, and are NOT_CREATED, ON, OFF, and ERROR. State transitions are shown in the diagram as process flow diagrams. A state transition is triggered by a signal (drawn as solid bars). Signals may originate from interrupts, a planning module, an external subsystem, or from the user through a graphical user interface. In response to a signal, a transfer is made between the local and global SVAR tables to read INCONSTS or INVARS, then one of the user-defined functions is called, followed by another transfer between the local and global SVAR tables, to write OUTCONSTS or OUTVARS. State variable transfers are shown as ovals, while the user-defined functions, which form the PBO, are shown as rectangles.

The PBO method that is called depends on the state of the process and the signal that is received. For example, if a PBO is in the ON state, and it receives a *wakeup* signal, then it will execute the *cycle* method and remain in the ON state. On the other hand, if the PBO is in the ON state, and receives the *kill* signal from an a configuration manager, then it will execute the *off* method, followed by the *kill* method, then enter the NOT-CREATED state.

The framework process, as shown, evolved over several years as we designed and tested many variations, in order to obtain a common program structure for all software components. The diagram represents the most recent revision in the evolution of the structure. The detailed PBO framework implies many design decisions, as described next

5.2.1 Notes about Framework Process

Despite the seeming complexity of the framework, dissecting it into pieces shows that it is indeed rather simple. In the steady state, PBO processes are all in the ON state, and executing their *cycle* method once per cycle or event, going back to sleep until the next *wakeup* signal. Note that the only difference between a periodic process and an aperiodic server is the source of the wakeup signal. For a periodic process, the wakeup signal is received from the timer. For the aperiodic processes, the process blocks on a semaphore, message, or event, as defined by the *sync* method of the PBO.

The autonomous nature of the PBO allows the most popular scheduling algorithms, such as the rate monotonic static priority [15], earliest-deadline-first [15], or maximum-urgency-first dynamic priority scheduling [39] algorithms, to be used to schedule PBOs. The control systems designer only needs to specify the frequency of the *cycle* routine for each PBO. Determining the optimal set of frequencies is a control systems issue, however, and beyond the scope of this paper.

As a result of the underlying timing error detection and handling mechanism built into the Chimera RTOS [36], aperiodic servers can use the same fundamental structure as periodic processes. The framework can define aperiodic processes as either deferable or sporadic servers, and use them with either the rate monotonic static priority or maximum-urgency-first dynamic priority scheduling algorithms to ensure predictable scheduling.

The remainder of the framework handles the initialization and termination, reconfiguration, and error handling for the PBO. To support dynamic reconfiguration, a two-stage initialization and termination is used. High-overhead

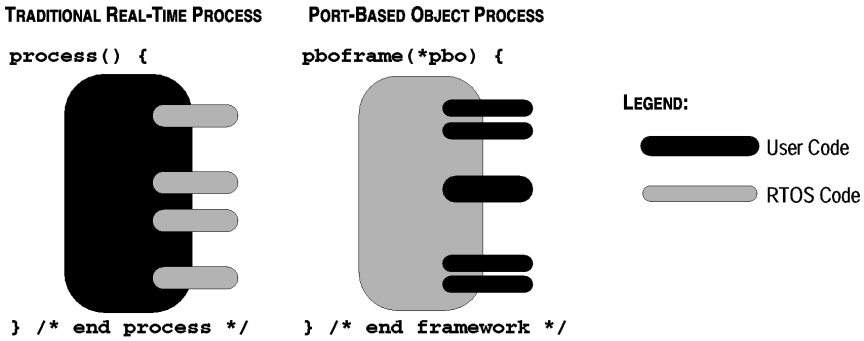


Fig. 9. Comparison of POSIX real-time processes and port-based object programming paradigms.

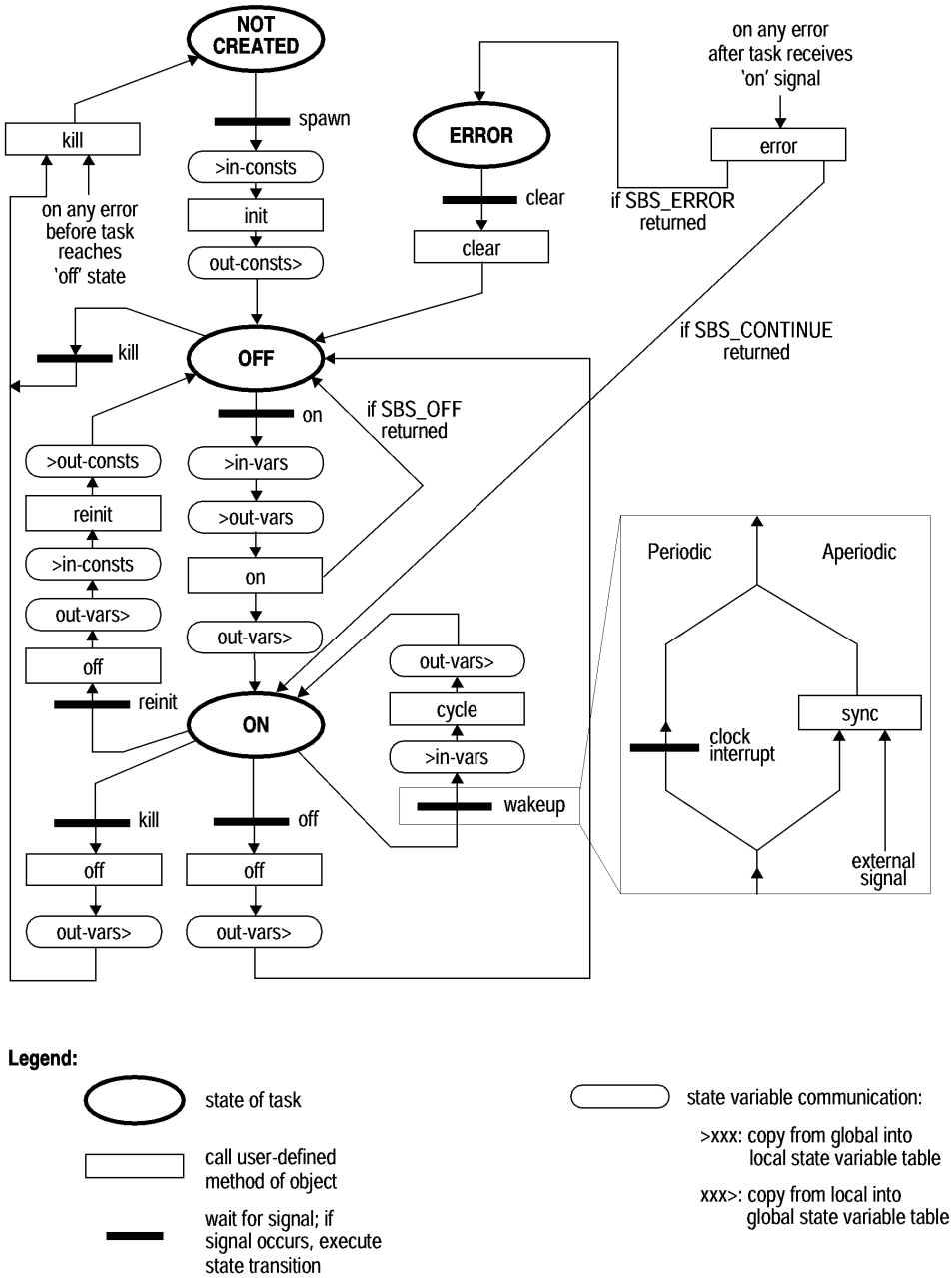


Fig. 10. Finite state machine for the port-based object framework process. State transitions are shown as process flow diagrams.

initialization of a new process can be performed upon system startup, or in the background, in preparation for being activated. The initialization includes creating a process's context, dynamically allocating its memory, creating a local table and translating I/O port symbols into pointers to the global table, and calling the user-defined *init* method. The process then waits in the OFF state, and can be viewed as being in a standby mode for a dynamic reconfiguration. When an *on* signal is received, the local table is updated to reflect the current state of the system, and execution begins.

Perhaps one of the keys to supporting dynamic reconfiguration is handling the initialization of the port variables. Before calling the *on* method, it is necessary to read the *OUTVARS* in addition to the *INVARS*. This solution resulted after many trials with initialization, trying to determine the best way to ensure that when a process is activated, its view of the environment is correct. Since the process is not previously executing, it is possible that some other process is generating those *OUTVARS*. In general, a process that outputs *OUTVARS* knows the same values on the subsequent cycle because they are in the local table. However, in a process' first cycle, this is not the case. Ultimately, rectifying the situation is a simple matter of also reading the *OUTVARS* during activation, thus updating the local table to properly reflect the state of the system. The *on* method of the process is then called, such that if necessary, these *OUTVARS* can be updated, before the process enters the ON state.

The time for a process P to be activated, $C_{on,P}$ is bounded, if the user-defined *on* method is bounded. It is,

$$C_{on,P} = t_{IP} + 2t_{OP} + C'_{on,P} + \Delta_{OS} \tag{8}$$

where t_{IP} and t_{OP} are defined in (3), $C'_{on,P}$ is the execution time of the user-defined *on* method of process P, and Δ_{OS} is the operating system overhead for sending a signal and performing a context switch. Typical values for $C_{on,P}$ in our system were in the order of (100 μsec + $C'_{on,P}$).

The deactivation of a process is similarly time-bounded. Thus, the time to perform a dynamic reconfiguration is the sum of the time to activate new processes and deactivate running processes that form the differential between the old and new configurations. Since it is generally possible to activate and deactivate several processes within a millisecond, it is possible to execute one controller module one cycle, followed by executing a different controller on the next cycle.

As a precautionary measure in case of transient overload during a dynamic reconfiguration, an *illegal configuration flag* is temporarily set, which indicates that not all required output is being produced. The framework provides this mechanism primarily for use of HD interface components that send output to actuators. The most critical processes in the system, which should have a priority higher than the processes being reconfigured, can test this flag, and if it is set, can choose to ignore its *INVARS*, and go into locally stable execution. For example, a robot interface module can select to keep the velocity of the joints constant, until the dynamic reconfiguration is complete, signaled by the illegal configuration flag being reset. Determining when it is safe to perform a dynamic reconfiguration is beyond the scope of the framework. The framework provides the mechanisms only. Developing policies that ensure stable execution during a reconfiguration is usually application specific. In our experiments, we use a conservative approach of ensuring that the robot is temporarily at rest (i.e., velocity and acceleration are both zero before dynamic reconfiguration begins). Further research is required in order to develop more aggressive policies.

Error detection and handling is implemented using a global error handling mechanism, as described in [37]. Whenever an error occurs, an error signal is generated, and a user-defined error handler is called. The framework automatically initializes this error handler to be the *error* method of the PBO. The purpose of the *error* method is to attempt automated recovery. If that fails, the process goes into the ERROR state, indicating that user intervention is required. Once the user attempts to fix the problem, a clear signal is generated. The clear method checks that the error was indeed fixed. If it was, then the process returns to the OFF state and can be reactivated; if not, the process remains in the ERROR state. See [14] for further research into error handling for PBOs.

Whenever one or more processes are in the ERROR state, the illegal configuration flag described above is set. This again indicates to critical modules that there exists at least one other module in the configuration not producing output as required, and therefore the *INVARS* of that process may not have been properly computed.

During a dynamic reconfiguration, a new process can be created in the background, such that it updates some of the configuration constants. Before this process can be activated, some of the running processes might need to be initialized, if they have configured themselves previously based on different *INCONSTS*. In this case, a *reinit* signal is sent to those processes that have the updated *OUTCONST* as an *INCONST*. The new process can only be activated after processes have reinitialized.

Statically configurable processes are subsets of dynamically reconfigurable processes. These processes go directly to the ON state after initialization. Therefore, if the OFF state is removed, and any transition emanating from the OFF state removed, the result is a framework process that can be used for statically configurable systems. In our applications requiring only static configurability, however, we still define the modules as dynamically reconfigurable, but send an *on* signal immediately after the *init* signal.

5.3 Coding a Port-Based Object

The structure of a PBO is designed so that a control engineer can simply define module-specific code, and not be concerned with any of the details of creating a real-time process. A template can be created for any specific PBO, given an *.rmod* file, as was shown in Fig. 2. The control engineer then only has to fill in the blanks, which is to define the methods of the PBO to perform the module-specific functionality. As an example, Fig. 11 shows the template and control engineer's code for the *tball* module. The regular font shows the template code for *tball*, given the information in Table 1, while the **bold** font shows code written by the control engineer.

```
/*** Module tball
[Note: Most Template Comments Deleted in this diagram]
*/
#include <chimera.h>
#include <sbs.h>
#include <math.h>
#include <iod.h>

typedef struct {
        /* Port Variables */
    float *Xdot ;
        /* Other Module-Specific Variables */
    float gain[6];
    IOD *tball;
    int finished;
} tballLocal_t;

    /* Start of Module-specific header Info */
#define GO_TO_FLOW(i) SBS_SIG(SBS_SIG_END+2+i)
    /* End of Module-specific header Info */
SBS_MODULE(tball);

int  tballInit(cfigInfo_t *cinfo, sbsTask_t *local, sbsTask_t *stask) {
    sbsSvar_t*svar = &stask->svar;

        /* Standard Template Code */
    local->Xdot=svarTranslateValue(svar->vartable, "X^_REF",float);
        /* Start of Module-specific 'init' code */ {
    char     cfigfile[MAXFNAMELEN];
    cfigCompulsory(cinfo, "GAINS", local->gain, CFIG_FLOAT, 6);
    cfigCompulsory(cinfo, "DEVICEFILE", cfigfile,
        CFIG_STRING, MAXFNAMELEN);

      /* Locate and initialize trackball device */
    local->tball = iodInit(cfigfile, 0);
    iodControl(local->tball, IOD_GAIN, local->gain);
        } /* End of Module-specific 'init' code */
    return (int) local;
}

int  tballReinit(tballLocal_t *local, sbsTask_t *stask) {
        /* Start of Module-specific 'reinit' code */
    /* Nothing to do here */
        /* End of Module-specific 'reinit' code */
  return I_OK;
}
```

```
int  tballOn(tballLocal_t *local, sbsTask_t *stask) {
        /* Start of Module-specific 'on' code */
    local->finished = 0;
        /* End of Module-specific 'on' code */
    return I_OK;
}

int  tballCycle(tballLocal_t *local, sbsTask_t *stask) {
        /* Template Variables */
    float*Xdot = local->Xdot;
        /* Start of Module-specific 'cycle' code */
    static double btime = 0.0;
    float val[6];
    int buttons[2];

    if (local->finished == 1)
        return I_OK;
    iodRead(local->tball, IOD_BUT, &buttons);
    if (buttons[0] == 7) {
        local->finished = 1;
        sbsSigSend(stask,SBS_SIG_FINISH | GO_TO_FLOW(0))
        kprintf("Finished signal sent, waiting to be OFFed");
        return I_OK;
    }
        /* Read the trackball values now. */
    iodRead(local->tball, IOD_GAIN, val);
    Xdot[0] = val[0];      Xdot[1] = –val[2];
    Xdot[2] = val[1];      Xdot[3] = val[3];
    Xdot[4] = –val[5];     Xdot[5] = val[4];
        /* End of Module-specific 'cycle' code */
    return I_OK;
}

int  tballOff(tballLocal_t *local, sbsTask_t *stask) {
        /* Start of Module-specific 'off' code */
    /* Nothing to do for this module */
        /* End of Module-specific 'off' code */
    return I_OK;
}
int  tballKill(tballLocal_t *local, sbsTask_t *stask) {
        /* Start of Module-specific 'kill' code */
    iodFinish(local->tball);
        /* End of Module-specific 'kill' code */
    return I_OK;
}
```

Fig. 11. Sample .c code for *tball* PBO. Template code uses regular font; user code is in **bold**.

The encapsulated data of the object is stored in the structure called *tballLocal_t*. Part of this structure is generated by the framework, to include pointers to the local SVAR table. The remainder of the structure is user-definable, so that the control engineer can place any "global" variables for their PBO. The data in this structure is made available to every method. The methods all have specific names, which is of the form *xxxYyyy*. *xxx* represents the module name, and *Yyyy* represents the method name, as was depicted in Fig. 10.

The rigid process structure provides strict guidelines for control engineers, telling them exactly where to put what kind of code. It removes any guesswork, reduces the amount of code they must write, and guarantees that synchronization and communication works from the beginning.

6 Discussion

While our DRRTS solution meets the requirements set forth in Section 1, we have since realized many other, and in many cases more important, potential benefits of the solution:

Rapid development through component-based design. Component-based design minimizes the amount of new code that must be written when an application is developed. A framework for DRRTS eliminates the need to write any "glue" code, thus going a step beyond reuse of modular components.

Hardware/software co-design. One form of system-level design of embedded applications uses a mixture of hardware and software components [10]. The co-design approach allows the hardware and DRRTS components to be tightly coupled throughout the design process.

Tele-configuration of services. Telecommunication companies, such as cable, satellite TV, and telephone, are increasingly transmitting control information over the data transmission medium. DRRTS components can be used to dynamically change the configuration, hence the services provided, or to perform remote upgrades and maintenance of existing services.

Evolutionary design. Complex systems may require continuous hardware and software upgrades during the lifetime of the system in response to technological ad-

vancements, environmental change, or alteration of system goals. DRRTS components are designed to undergo such evolution, as individual modules can be replaced incrementally and independently.

Flexibility for fine-tuning after implementation. A DRRTS framework offers considerable flexibility for fine tuning an application to "make it work." It can have reconfiguration options such as switching between static and dynamic scheduling algorithms, using time-based decomposition to improve CPU utilization, and easily converting interrupt handlers to aperiodic servers and vice versa.

Increased reliability through automated analysis. The internal structure of a DRRTS component is well defined, based on a theoretical model. This structure allows for the automation of such things as performance measurement, configuration verification, and scheduling analysis.

These additional benefits fuel our current research effort into developing advanced RTOS technology for supporting dynamically reconfigurable systems.

REFERENCES

[1] B. Abbott et al., "Model-Based Software Synthesis," *IEEE Software*, vol. 10, no. 3, pp. 42-52, May 1993.

[2] J.M. Adan and M.F. Magalhaes, "Developing Reconfigurable Distributed Hard Real-Time Control Systems in STER," *Algorithms and Architectures for Real-Time Control, Proc. IFAC Workshop*, pp. 147-152, Oxford: Pergamon Press, Sept. 1991.

[3] J.S. Albus, H.G. McCain, and R. Lumia, *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*, Technical Note 1235, *Nat'l Inst. of Standards and Technology*, Gaithersburg, Md., Apr. 1989.

[4] B.W. Beach, "Connecting Software Components with Declarative Glue," *Proc. Int'l Conf. Software Eng.*, pp. 11-15, Melbourne, Australia, IEEE Press, 1992.

[5] T.E. Bihari and P. Gopinath, "Object-Oriented Real-Time Systems: Concepts and Examples," *Computer*, vol. 25, no. 12, pp. 25-32, Dec. 1992.

[6] B.A. Blake and P. Jalics, "An Assessment of Object-Oriented Methods and C++," *J. Object-Oriented Programming*, vol. 9, no. 1, pp. 42-48, Mar.-Apr. 1996.

[7] R.C. Dorf, *Modern Control Systems*, third edition. London: Addison-Wesley, 1980.

[8] M.W. Gertz, D.B. Stewart, and P.K. Khosla, "A Human Machine Interface for Distributed Virtual Laboratories," *IEEE Robotics and Automation Magazine*, vol. 1, no. 1, pp. 5-13, Dec. 1994.

[9] *IV3230 VMEbus Single Board Computer and MultiProcessing Engine User's Manual*, Computer Systems Division, Ironics Inc., Ithaca, New York, 1991.

[10] A. Kalavade, "System-Level Codesign of Mixed Hardware-Software Systems," doctoral dissertation, Univ. of California, Berkeley, 1995.

[11] L. Kelmar and P.K. Khosla, "Automatic Generation of Forward and Inverse Kinematics for a Reconfigurable Modular Manipulator System," *J. Robotics Systems*, vol. 7, no. 4, pp. 599-619, Aug. 1990.

[12] D. Kinny, M. Georgeff, and A. Rao, "A Methodology and Modeling Technique for Systems of BDI Agents: Agents Breaking Away," *Lecture Notes in Artificial Intelligence LNAI 1038*. Berlin: Springer-Verlag, 1996.

[13] D.A. Lamb, "IDL: Sharing Intermediate Representations," *ACM Trans. on Programming Languages and Systems*, vol. 9, no. 3, pp. 297-318, July 1987.

[14] J. Lang, "A Distributed and Time-Bounded Exception Handling Mechanism for Dynamically Reconfigurable Real-Time Software," masters thesis, Dept. Electrical Engineering, Univ. of Maryland, College Park, 1997.

[15] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *J. ACM*, vol. 20, no. 1, pp. 44-61, Jan. 1973.

[16] D.M. Lyons and M.A. Arbib, "A Formal Model of Computation for Sensory-Based Robotics," *IEEE Trans. Robotics and Automation*, vol. 5, no. 3, pp. 280-293, June 1989,.

[17] J. Magee, N. Dulay, and J. Kramer, "Structuring Parallel and Distributed Programs," *IEE Software Eng. J.*, vol. 8, no. 2, pp. 73-82, Mar. 1993.

[18] J. Magee et al., "Configuring Object-Based Distributed Programs in REX," *IEE Software Eng. J.*, vol. 8, no. 2, pp. 139-149, Mar. 1992.

[19] J. Magee, J. Kramer, and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Trans Software Eng.*, vol. 15, no. 6, pp. 663-675, 1989.

[20] L.D. Molesky, C. Shen, and G. Zlokapa, "Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems," *The J. Real-Time Systems*, vol. 2, no. 3, pp. 163-180, Sept. 1990.

[21] D.L. Parnas, P.C. Clements, and D.M. Weiss, "The Modular Structure of Complex Systems," *IEEE Trans. Software Eng.*, vol. 11, no. 3, pp. 259-266, Mar. 1985.

[22] J. Purtilo, "The Polylith Software Bus," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 1, pp. 151-174, Jan. 1994

[23] J.M. Purtilo and J.M. Atlee, "Module Reuse by Interface Adaptation," *Software: Practice and Experience*, vol. 21, no. 6, pp. 539-556, June 1991.

[24] R. Rajkumar, "Real-Time Synchronization Protocols for Shared Memory Multiprocessors," *Proc. 10th Int'l Conf. Distributed Computing Systems*, pp. 116-123, Paris, 1990.

[25] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers*, vol. 39, no. 9, Sept. 1990.

[26] S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proc. Int'l Conf. Application Specific Array Processors*, Berkeley, Calif., pp. 4-7, 1992.

[27] S.R. Schach, *Software Eng.*, second edition. Asken Associates, 1993.

[28] D.E. Schmitz et al., "CHIMERA: A Real-Time Programming Environment for Manipulator Control," *IEEE Int'l Conf. Robotics and Automation*, Phoenix, Ariz., pp. 846-852, 1989.

[29] D.E. Schmitz, P.K. Khosla, and T. Kanade, "The CMU Reconfigurable Modular Manipulator System," *Proc. Int'l Symp. and Exposition on Robots* (designated 19th ISIR), Sydney, Australia, pp. 473-488, 1988.

[30] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons, 1994.

[31] T.E. Smith and D.E. Setliff, "Towards an Automatic Synthesis System for Real-time Software," *Proc. Real-Time Systems Symp.*, San Antonio, Texas, pp. 34-42, 1991.

[32] M. Steenstrup, M.A. Arbib, and E.G. Manes, "Port Automata and the Algebra of Concurrent Processes," *J. Computer and System Sciences*, vol. 27, no. 1, pp. 29-50, Aug. 1983.

[33] W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured Design," *IBM Systems J.*, vol. 13, no. 2, pp. 115-139, 1974.

[34] D.B. Stewart and G. Arora, "Dynamically Reconfigurable Embedded Systems–Does it Make Sense?" *Proc. Real Time Application Workshop*, Montreal, Oct. 1996.

[35] D.B. Stewart and P.K. Khosla, "Policy-Independent RTOS Mechanisms for Timing Error Detection, Handling, and Monitoring," *Proc. IEEE High Assurance Systems Eng. Workshop*, Niagara, Ont. Canada, Oct. 1996.

[36] D.B. Stewart and P.K. Khosla, "Mechanisms for Detecting and Handling Timing Errors," *Comm. ACM*, vol. 40, no. 1, pp. 87-94, Jan. 1997.

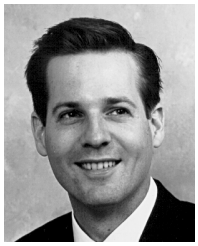
[37] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, "The Chimera II Real-Time Operating System for Advanced Sensor-based Control Applications," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 22, no. 6, pp. 1,282-1,295, Nov./Dec. 1992.

[38] D.B. Stewart and P.K. Khosla, "Chimera 3.1: The Real-Time Operating System for Reconfigurable Sensor-Based Control Systems," program documentation, Advanced Manipulators Laboratory, The Robotics Inst. and Dept. Electrical and Computer Eng., Carnegie Mellon Univ., Pittsburgh, http:// www.ee.umd.edu/serts/bib/ manuals/Chimera.html

[39] D.B. Stewart, "Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems," doctoral dissertation, Carnegie Mellon Univ., Dept. Electrical and Computer Eng., Pittsburgh, 1994, http://www.ee.umd.edu/serts/bib/thesis/dstewart.html.

[40] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, "CHIMERA II: A Real-Time Multiprocessing Environment for Sensor-Based Robot Control," *Proc. IEEE Int'l Symp. Intelligent Control*, Albany, N.Y., pp. 265-271, Sept. 1989.

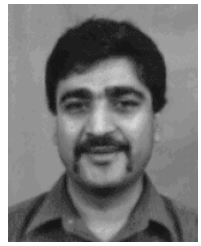
[41] *VBT-325 The VME+ Analyzer System User's Manual*, VMETRO Inc., Houston, 1995.
[42] P. Wegner, "Dimensions of Object-Oriented Programming," *Computer*, vol. 25, no. 10, pp. 12-20, Oct. 1992.
[43] P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," *OOPS Messenger*, vol. 1, no. 1, pp. 7-84, Aug. 1990.



David B. Stewart received his MS degree (1989) and his PhD degree (1994) in electrical and computer engineering from Carnegie Mellon University. He received a BEng degree (1988, great distinction) in computer engineering from Concordia University. Dr. Stewart is an assistant professor with the Department of Electrical Engineering at the University of Maryland, College Park. He has a joint appointment with the University of Maryland Institute for Advanced Computer Studies (UMIACS) and is a faculty affiliate with the Institute for Systems Research (ISR). He is director of the Software Engineering for Real-Time Systems (SERTS) Laboratory. His research lies in the areas of advanced real-time operating systems technology and component-based software for embedded systems. In 1991, he was a visiting researcher at the Jet Propulsion Laboratory, California Institute of Technology.

In May 1997, Dr. Stewart received the George Corcoran Award in recognition of teaching and education leadership at the University of Maryland, College Park Campus, effective contribution at the national level, and creative and other scholarly activities related to electrical engineering education. He is a recipient of the Natural Sciences and Engineering Research Council of Canada 1967 Science and Engineering Scholarship. He has also been awarded the Chait Medal and Computer Engineering Medal from Concordia University; the Prize of Excellence from the Professional Order of Engineer's of Quebec, Canada; the Myer F. Pollock Scholarship for academic excellence; the Concordia University Entrance Scholarship; the Independent Order of Forester's University Entrance Scholarship; the National Honor Society of Canada Scholarship; and the Josten's Foundation Scholarship. Dr. Stewart is a member of the IEEE.

Richard A. Volpe received his MS degree (1986) and his PhD degree (1990) in applied physics from Carnegie Mellon University, where he was a U.S. Air Force Laboratory graduate fellow. His thesis research concentrated on real-time force and impact control of robotic manipulators. In December 1990, he became a senior member of the technical staff at the Jet Propulsion Laboratory, California Institute of Technology. Until late 1993, he was a member of the Remote Surface Inspection Project, investigating sensor-based control technology for telerobotic inspection of the International Space Station. After late 1993, he led the development of Rocky 7, a next generation mobile robot prototype for extended-traverse sampling missions on Mars. Recently he received a NASA Exceptional Achievement Award for this work, and has joined the spaceflight team for this 2001 rover mission. His research interests include real-time sensor-based control, robot design, path planning, and computer vision. Volpe is a member of the IEEE.



Pradeep K. Khosla received the BTech degree (honors) from IIT (Kharagpur, India), and both MS and PhD degrees from Carnegie Mellon University. He is currently professor of electrical and computer engineering and robotics, and founding director of the Institute for Complex Engineered Systems (which includes the former Engineering Design Research Center—an NSF ERC) at Carnegie Mellon University. Prior to joining Carnegie Mellon, he worked with Tata Consulting Engineers and Siemens in the area of real-time control. From January 1994 to August 1996, he was on leave from Carnegie Mellon and served as a DARPA program manager in the Software and Intelligent Systems Technology Office (SISTO), Defense Sciences Office (DSO), and Tactical Technology Office (TTO), where he managed advanced research and development programs in the areas of Information based design and manufacturing, web based Information technology infrastructure, real-time planning, real-time software, sensor-based control, and robotics.

Professor Khosla's research has focused on the theme of "creating complex electro-mechanical and information systems through composition of and collaboration amongst building blocks." Specifically, his interests are in the area of integrated design-manufacturing systems, collaborating robots, agent-based architectures for design and control, software composition for real-time systems, reconfigurable autonomous systems, and human systems interface. He is particularly interested in the application domains of design and manufacturing, and unstructured environments. He is involved in electrical and computer engineering, design, and robotics education, both at the graduate and the undergraduate level. He was a member of the committee that formulated a curriculum for the PhD program in robotics at Carnegie Mellon. He was also a member of the Wipe the Slate Clean Committee that created a new four year undergraduate ECE degree curriculum at CMU. In support of the new curriculum, he developed and implemented the notion of virtual laboratory, and an introductory freshman level course "Introduction to Electrical and Computer Engineering" that emphasizes the notion of teaching in context.

Professor Khosla was the program vice-chair for the 1989 IEEE International Conference on Systems Engineering; general chair for the 1990 IEEE International Conference on Systems Engineering; program vice chair of the 1993 International Coference on Robotics and Automation; general co-chair of the 1995 Intelligent Robotics Systems (IROS) conference; and program vice-chair for the 1997 IEEE Robotics and Automation Conference. He has served as a member of the AdCom of the IEEE Systems, Man and Cybernetics Society, technical editor of the *IEEE Transactions on Robotics and Automation*, and cair of the education committee of the IEEE Robotics and Automation Society.

Professor Khosla is a recipient of the Inlaks Fellowship, United Kingdom; the Carnegie Institute of Technology Ladd award for excellence in research in 1989; two NASA Tech Brief awards; and was elected as an IEEE fellow in January 1995. Professor Khosla's research has resulted in two books and more than 200 journal articles, conference papers, and book contributions. Professor Khosla served as a member of the technical advisory board of the Next Generation Controller Project (WPAFB, U.S. Air Force and Martin Marietta), RIMCC, and the Sample Acquisition Analysis and Preservation Project (Jet Propulsion Laboratory, NASA). He has been an invited participant to the Department of Commerce workshops on the Intelligent Manufacturing Systems program, and USA-Japan R&D consortia and collaboration. He is a consultant to several industries in the USA and a co-founder and chairman of the board of K2T inc.—a high tech company based in Pittsburgh.