Draft of article which appears in
Intl. Journal of Software Engineering and Knowledge Engineering
v.6, n.2, pp. 249-277, June 1996.

THE CHIMERA METHODOLOGY: DESIGNING DYNAMICALLY RECONFIGURABLE AND REUSABLE REAL-TIME SOFTWARE USING PORT-BASED OBJECTS[1]

David B. Stewart
Electrical Engineering Department
University of Maryland
College Park, MD 20742
dstewart@eng.umd.edu

Pradeep K. Khosla
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213
pkk@ri.cmu.edu

Abstract: *The Chimera Methodology is a software engineering paradigm that enables rapid development of real-time applications through use of dynamically reconfigurable and reusable software. It is targeted towards a distributed shared memory computing environment. The primary contribution of this research is the port-based object model of a real-time software component. The model is obtained by applying the port-automaton formal computational theory to object-based design. A finite state machine, detailed interface specifications, and a C-language template are used to define the port-based object. Tools to support the integration, scheduling, and state variable communication between the objects have been developed and incorporated into the Chimera Real-Time Operating System. Techniques for verifying correctness and analyzing performance are also provided for configuration managers that integrate software designed using the port-based object model.*

1. INTRODUCTION

The Chimera Methodology is a software engineering paradigm that enables rapid development of real-time applications through use of dynamically reconfigurable and reusable software. It is targeted towards a distributed shared memory computing environment, and implementation has been on a set of single board computers on a VMEbus, hosted by a Sun SPARCstation. The Chimera name is used since this methodology cultivated from the Chimera Real-Time Operating System Project [39].

As an example of the current state of real-time software development, consider the development of a real-time application where a specific software algorithm developed at another site is required. The software

[1] The research presented in this paper is supported, in part, by the Electrical Engineering Dept. and Institute for Advanced Computer Studies at University of Maryland, the Electrical and Computer Engineering Dept. and the Robotics Institute at Carnegie Mellon University, Sandia National Laboratories, and NASA. Partial funding for D. Stewart was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a graduate fellowship

designer must first go to the library or search through a database of abstracts to find a book or journal article describing the mathematical theory or computer science algorithm they are seeking. After a copy of the book or paper is obtained, the article is read closely and the perhaps sketchy details of the desired algorithm deciphered. Next, the algorithm is implemented, for which code must be written, tested, and debugged. When that is completed, even more code must be written to integrate the new module into the final application, followed by further testing, debugging, and ultimately fine-tuning of timing parameters. This process can easily take days or weeks of development time for each algorithm needed in an application, and thus may result in a software development cycle that is several months long for a single real-time application.

An alternative to the current development method is to use software assembly to reduce software development time. Software assembly is a visual programming technique that can be used to rapidly create reconfigurable systems. It requires that the underlying software be decomposed into two parts: 1) the configuration management tool and 2) well defined and structured reusable software modules. Applications are created by configuring the software modules using the configuration management tool.

For example, software designers who need a specific algorithm search a global distributed software library. The library can be based on the hypermedia information system currently available with the World-Wide Web [44]. When a suitable algorithm is found, they not only get the written theory, but they can also follow a link to a reusable software module created by the original authors. The algorithm is already programmed, fully tested, and debugged. With an action as simple as a mouse-click, the software component is copied into the designer's personal or project library. It is then ready to be used in a custom application. This process would take a few minutes at most. A complete application is created by putting together these software building-blocks using the configuration management tool, and allowing an underlying real-time operating system (RTOS) to handle the timing and communication. Within hours, a complete application could be assembled, as compared to the months that it would take to create the application using conventional software design methods.

Until now, the above idealistic programming environment has been unrealizable, because transfer, reuse, and integration of real-time software is difficult and often seemingly impossible. First, A lack of software interfacing standards and the use of specialized and incompatible hardware results in low software portability. Second, the glue code to integrate software modules created at different sites is often difficult to write and debug. Third, most real-time systems are still developed using trial-and-error techniques, since getting the timing and resource sharing correct for the application is non-trivial.

These issues have been addressed in the evolution of the Chimera Methodology. Software models are provided which can be used as the basis for interfacing standards, code templates, and creating reconfigurable device drivers. The port-automaton theory has been applied to object design in order to minimize the dependencies between real-time objects and improve their schedulability using popular real-time scheduling algorithms. Object-management services and communication mechanisms have been incorporated into the Chimera RTOS to automatically integrate software components, in order to eliminate the need for configuration languages or writing and debugging glue code. Predictability of an assembled application can be analyzed and verified, using techniques that can be incorporated into the configuration manager.

The remainder of this paper is organized as follows: Background and related work into developing reconfigurable software is discussed in Section 2. The port-based object software model which forms the basis of the Chimera Methodology is described in Section 3. Techniques for configuration analysis and verification are presented in Section 4. Examples of Onika, a visual programming configuration manager based on the Chimera Methodology, are given in Section 5. Finally, the paradigm is summarized in Section 6.

2. Background and Related Work

Reconfigurable systems can be categorized as either statically configurable or dynamically reconfigurable. A statically configurable system is one where reusable software modules are selected and configured off-line, and only executed after configuration is complete. In contrast, dynamically reconfigurable systems can be modified on-line, without the need to recompile the application nor shutdown and reboot the system. Changes to the application can come in the form of updating the hardware configuration, changing the set of software modules executing, or modifying execution parameters of either the entire application or a selected subset of modules. This section describes various approaches to developing reconfigurable systems.

Software synthesis, also known as automatic code generation, generally employs artificial intelligence techniques such as knowledge bases [1] [36] and expert systems [16] [31] to generate the glue code for integrating reusable software components. Because of the need to regenerate and recompile the code each time a reconfiguration is required, this method is only suitable for statically configurable systems.

Interface adaptation involves modifying the interfaces of software modules based on the other software modules with which they must communicate in order to obtain the required software integration. In these systems, an interface specification language is used to provide a general wrapper interface and to allow meaningful data to be interchanged between the modules [14] [20] [24]. This method has led to the notion of a software bus, where an underlying server or transport mechanism adapts to the software module, rather than having the software modules adapt to the transport mechanism [3] [29]. Interface adaptation is targeted at re-engineering software that was not originally designed to be reused or reconfigured. Most of these methods only support static configuration, and none of them support real-time execution.

A graphical approach to developing reconfigurable systems is software assembly. LabView is a commercial software assembly tool that supports dynamic reconfigurability [42]. Although Labview provides an ideal environment for rapidly creating data acquisition and analysis applications, it does not support multi-threaded design required for creating control systems and distributed applications.

The Regis environment [22] is a follow-up to the REX system [24] and uses the configuration management/program structure separation to obtain dynamically reconfigurable software. Regis uses the Darwin configuration language, based on the Conic[23] interface adaptation method to structure the components using input and output communication objects. ConicDraw [18] can then be used to assemble the software graphically. In the Regis environment, communication is obtained through message passing in order to support concurrency in a distributed network-based environment.

The Kernel Tool Kit (KTK) [12] provides an alternate approach to software assembly to obtain dynamic reconfigurability. KTK is an object-based operating system that provides kernel support for a dynamically reconfigurable distributed network environments. The authors have shown that this method can be used to reconfigure a parallel application to improve overall performance. However, because this approach does not have the configuration management/program structure separation it does not support software assembly.

The aforementioned research does not address any of the real-time issues associated with designing reconfigurable systems. The MetaH configuration language [43] does address real-time issues associated with developing reconfigurable guidance, navigation, and control. However, the MetaH real-time modelling and analysis does not support multiprocessing nor dynamic reconfigurability.

The Chaos Real-Time Operating System [33] was designed to use object-oriented design with real-time systems in order to provide reconfigurability. However, an object-oriented programming language generally performs run-time dynamic binding to support this inheritance, and objects of different classes communicate with each other through messages, where the message invokes the method of another object [6]. Such dynamic binding and message passing creates unpredictable execution delays, especially in a distributed environment, and as a result is not suitable for the design of real-time systems. Chaos addresses the dynamic binding issue by performing static binding during the compilation and linking stages, thus allowing for predictable execution of the real-time application. The Chaos system addresses the real-time message passing issue by creating a variety of specialized messages which are tailored to the target application. As stated by the authors of Chaos in [4], this, to some extent, ruins the object model's uniformity and partially defeats the purpose of using the object-oriented methodology for developing real-time systems. It also prevents Chaos from being used for dynamically reconfigurable systems.

Meta [26] is a tool for instrumentizing software components to enable distributed configuration management in a dynamically reconfigurable system. It uses a software concept of *sensors* and *actuators* to better monitor and modify configurations to ensure correctness and add robustness during dynamic reconfiguration. As discussed in Section 3.6, this work could potentially be used in conjunction with the Chimera Methodology to improve the robustness of a subsystem's signalling mechanisms.

The Chimera Methodology addresses the design of dynamically reconfigurable real-time systems which are implemented in a distributed shared memory environment. Although many distributed applications are implemented over local-area networks, standard Internet protocols are not suitable for real-time systems. As a result, multiprocessor real-time applications tend to be implemented in a distributed shared memory environment. A popular hardware architecture providing this environment is the VMEbus. However, with the increasing availability of multiprocessor workstations with real-time operating systems (such as the Sun SPARCstation MP514 with Solaris 2.x)[19], the concepts developed with the Chimera Methodology can form the basis for supporting dynamically reconfigurable software in these workstation environments. The Chimera Methodology combines object-based design with the port-automaton computational model, as described next, to model dynamically reconfigurable real-time software components.

Streenstrup and Arbib [41] formally defined a concurrent process as a *port automaton*, where an output response is computed as a function of an input response. The automaton executes asynchronously and independently, and whenever input is needed, the most recent data available is obtained. The automaton may have internal states; however all communication with other concurrent processes are through the ports. The port-automaton theory was first applied to robotics by Lyons and Arbib [21], who constructed a special model of computation based on it, which was called *Robot Schemas*. The schema used the port-automaton theory to formalize the key computational characteristics of robot programming into a single mathematical model.

Arbib and Ehrig extended the work on robot schemas for algebraically specifying modular software for distributed systems by using *port specifications* to link modules [2]. The specification presented requires that there be exactly one input for every output link, and vice versa. The specification does not include any notion of objects nor any method to obtain reusability of the modules and reconfigurability of a subsystem, and they do not specify the mechanisms required to implement their model.

In our research, these port specifications have been combined with object-based design in order to create a model for reconfigurable real-time software components. The port specifications are also extended so that an output port can be spanned into multiple inputs and multiple outputs can be joined into a single input. In addition, operating system services are provided such that the communication through these ports can be performed in real-time and a set of port-based objects can be reconfigured dynamically. Through a joint effort with The Robotics Institute at Carnegie Mellon University, the Chimera Methodology was applied to several robotics applications. Most examples given in this paper are excerpts from those applications. In the next section, the port-based object software model is presented.

3. PORT-BASED OBJECTS

The *port-based object* is a new abstraction for real-time software components that applies the port automaton theory to object-based design. A port-based object has all the properties associated with standard objects, including internal state, code and data encapsulation, and characterization by its methods. It also has input, output, and resource ports for real-time communication. Input and output ports are used for integrating objects in the same subsystem, while resource ports are used for communication external to the subsystem, such as with the physical environment, a user interface, or other subsystems.

A link between two objects is created by connecting an output port of one module to a corresponding input port of another module, using port names to perform the binding. A configuration can be legal only if every input port in the system is connected to exactly one output port. A single output may be used as input by multiple tasks. In our diagrams, we represent such fanning of the output with just a dot at the intersection of multiple links, as shown in Figure 2. Both modules A and B require the same input p, and therefore the module C fans the single output p into two identical outputs, one for each A and B.

If two modules have the same output ports, then a join connector is required to merge the data into a single unambiguous output port, as shown in Figure 3. The join connector is a port-based object whose function is some kind of combining operation, such as a weighted average. In this example modules A and B are both generating a common output p. In order for any other module to use p as an input, it must only connect to a single output p. The configuration manager can modify the output port names of modules with the same outputs using the aliasing features provided by the underlying real-time operating system (RTOS) [37], such that they are two separate, intermediate variables. In our example, the output of module A becomes p', and the output of module B becomes p''. The join connector takes p' and p'' as inputs, and produces a single unambiguous output p.

A task is not required to have both input and output ports. Some tasks instead receive input from or send output to the external environment or to other subsystems, through the resource ports. Other tasks may generate data internally or receive data from an external subsystem (e.g. trajectory generator and vision subsystem interface) and hence not have any input ports, or just gather data (e.g. data logger and graphical display interface), and hence have no output ports. Any communication protocol can be used for the resource ports. This allows the hardware dependencies of an application to be encapsulated within a single port-based object, and forms the basis for developing reconfigurable device drivers, as discussed in Section 3.5. Note that time is always available to all tasks as a global value, and therefore tasks which required knowledge of the current time do not have to explicitly include it as an input port.

3.1 Integrating Port-Based Objects using State Variables

A task set is formed by linking multiple objects together to form either an open-loop or closed-loop subsystem. Each object in the subsystem executes as a separate task on one of the processors in a multiprocessor environment. An example of a fairly simple closed-loop subsystem is the PID joint control of a robot, as shown in Figure 4. It uses three modules: the *joint position trajectory generator*, the *PID joint position controller*, and the *torque-mode robot interface*.

The port-automaton computational model states that every task executes autonomously. At the beginning of each cycle, the task obtains the most recent data available from its input ports. At the end of the cycle, after performing any necessary computations, the task places new data onto its output ports. The task is completely unaware of the source and destination of the input and output data respectively.

Autonomous execution is desirable because it allows a task to execute independently of other tasks, and therefore does not block because another task is using a shared resource. Without blocking, the scheduling complexity is minimized and processor utilization optimized.

Tasks are independent, and therefore each task may execute at any frequency. The frequency of a task is typically defined by the control systems engineer. For example, the frequency can be increased to improve sampling rates and control system stability, it can be reduced to reduce processor utilization requirements, or it can be based on the speed of hardware such as sensors and actuators which the object is reading or writing respectively.

The port-automaton model assumes that the most recent data is always present at the input ports. Message queues do not satisfy this constraint. With message queues, it is possible that no messages are waiting, which occurs when a task producing an output is slower than the task requiring the data as input. On the other hand, if the task producing output is faster, then there is the possibility of multiple messages waiting at the port, and the next message to be received is not the most recent data. Messages create further problems if an output must be fanned into multiple inputs. In such cases, a message must be replicated, thus making the output data a function of the number of external tasks requiring the data. These characteristics of message queues contradicts the port-automaton model where an object is autonomous and unaware of its external environment.

State variables provide a superior alternative to messages for inter-object communication, as they do not have the problems associated with using message queues. Furthermore, they make use of shared memory, the fastest method of transferring data. A subsystem state is implemented by defining each port as a state variable. Writing to an output port then translates into updating the state variable, while reading from an input port translates into reading the state variable.

Since a state variable is shared, proper synchronization is required to ensure that only complete sets of data are read and written. A state variable can be a vector or other complex data structure, thus the entire transfer must be performed as a critical section. Using semaphores or similar types of synchronization violates the port-automaton model because they create dependencies between tasks. They introduce the possibility of priority inversion and deadlocks, and also increase the complexity of real-time scheduling analysis by adding blocking terms to the computations [35]. We now present our solutions for obtaining the required synchronization while maintaining an autonomous execution model.

3.2 State Variable Communication for Single Processor Environments

For single-processor environments, the synchronization can be obtained by locking the CPU, assuming that the size of a state variable transfer is small. Some people may argue that locking the CPU leads to possible missed deadlines or priority inversion. This would be true in the ideal case where a CPU has no operating system overhead. 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 a system call such as a full context switch [39]. If the total time that a CPU is locked in order to transfer a state variable is less than the worst-case locking of the microkernel due to operating system functions, then there is no additional effect on the predictability of the system. Only the worst-case execution time of that task must be increased by the transfer time, and that can be accounted for in the scheduling analysis.

An assumption that was made is that in most sensor-based control applications, the volume of data stored in by state variable is small. To justify this assumption, consider the example of the PID controller in Figure 4. Each state variable requires *ndof* transfers, where *ndof* is the number of degrees-of-freedom for the robot. A typical value for *ndof* is less than 10, and therefore the longest CPU locking for a state variable would be the time to perform 10 transfers. This would typically take less than 10 μsec on a CPU with a 100 μsec context switch time, considering that a context switch may contain as many as 100 operations for saving and restoring registers and updating a process control table.

One notable exception in which the assumption does not hold is for images. Vision applications can easily require several megabytes of data per second. In our model, such applications are implemented as a separate subsystem using special image processing hardware, and interfaced to the port-based objects using a reconfigurable device driver. For a vision subsystem, configurable inter-object communication can be implemented using high-volume data streams and synchronized tasks [13], instead of states and asynchronous tasks as described in this paper. Synchronous systems are more limiting because all tasks must execute at the same frequency and dynamic reconfigurability of more than one task at a time is usually not possible. However, synchronous systems do have an advantage for vision systems where a synchronized software pipeline is desired. The output of such a pipeline is typically a list of features or specific data points within an image. This low-volume output can then be sent to a control subsystem which uses the Chimera methodology, in order to incorporate vision into control applications [28].

Unfortunately, this simple mechanism cannot be used in multiprocessor environments, since locking only one of the CPUs will not provide the necessary atomic execution, and locking all the CPUs is not feasible.

3.3 State Variable Communication for Multiprocessor Environments

For multiprocessor environments, a global state variable table mechanism has been developed for port-based communication [40]. It is founded upon the combined use of global and local memory for the exchange of data between objects, as shown in Figure 5. A *global state variable table* is stored in shared memory. The variables in this table are a union of the input port and output port variables of all the objects that can be configured into the subsystem. Tasks corresponding to each control module cannot access this table directly. Instead, every task has its own local copy of the table, called the *local state variable table*. Only the variables used by the task are kept up-to-date in the local table. Since each task has its own copy of the local table, mutually exclusive access to it is not required. Therefore, a task can execute autonomously since it never has to lock the local table. The key is then to ensure that the local and global tables are updated to always contain the most recent data, and that the local table is never updated while a task is using the table.

Multiprocessor synchronization based on this type of shared memory architecture has been addressed in [30]. The *shared memory protocol* (SMP) is presented as an extension of the single processor priority ceiling protocol [35]. 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.

There are several problems which prevent the use of SMP within our framework. First, this method assumes that the local scheduling on each processor is the rate monotonic algorithm, with static priorities. As discussed in [37] it is desirable to use mixed or dynamic priority algorithms for scheduling reconfigurable systems, for which the protocol is not suitable. Second, 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 which is not under the control of software. Therefore the time to wait for the bus can be significant. Third, there is significant overhead associated with implementing SMP, which prevents its use with control applications requiring frequencies over 1000Hz. 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 [27]. 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.

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, which is 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 4.3 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 that only has fixed priority bus arbitration.

3.4 Detailed Port-Based Object Model

Traditionally, software modules are implemented as complete entities, which can invoke RTOS services through the use of system calls. Such an implementation model, however, forces each module to be responsible for its own communication and synchronization, and hence its integration with other modules.

Chimera instead uses an “inside-out” method of programming. Rather than the software modules invoking the RTOS whenever an operating system service is required, the RTOS framework is always executing, and it invokes methods of the port-based object as needed. Programmers who create software modules only have to define the methods of the port-based object; they do not have to write any kind of synchronization, communication, or other glue code. As a result, the creation of a reusable software component using the Chimera methodology is simpler than creating a traditional software module. The Chimera programming paradigm is also desirable because the operating system controls progress of every task, and as a result enables automatic execution time profiling and allows for the detection and handling of timing failures [37].

The detailed model of a port-based object is shown in Figure 6 as a finite state machine, with state transitions expanded as process flow diagrams. The ellipses show the possible states of the task, which can be *NOT-CREATED*, *OFF*, *ON*, or *ERROR*. The *OFF* state is for a task that has been created but is suspended while waiting for a start signal. The *ON* state represents a task that is ready to execute its next cycle, either in response to a timer wakeup signal for a periodic task, or the arrival of an event in the case of an aperiodic server. The *ERROR* state is for tasks that have encountered unrecoverable errors during their execution.

State transitions occur when a signal is received from an external subsystem, the configuration manager, or in the case of a wakeup signal, from the underlying hardware. These signals are shown in Figure 6 as solid bars. Methods of an object are invoked by the RTOS in response to these signals. Every object has each of the following special methods: *init, on, cycle, off, kill, error*, *clear, reinit* and *sync,* which are shown as rectangular boxes. Details of these methods are given below.

Immediately before and after each method of a port-based object is executed, a transfer is made between the local and global state variable tables. This ensures that a method always uses the most up-to-date data, and that new data is immediately placed into the global table. These state variable table transfers are shown in Figure 6 as oval boxes.

A port-based object can have two kinds of input: constant input (*in-const*) that is read only once during initialization; and variable input (*in-vars*) which is read at the beginning of each cycle for periodic tasks, or at the start of event processing for aperiodic tasks. Similarly, a task can have output constants (*out-const*) or output variables (*out-var*). Constants and variables are both transferred through the state variable table.

State constants are used in the creation of reconfigurable device drivers and generic software, as described in Section 3.5. The use of *in-consts* and *out-consts* by the modules create a necessary order for initialization of tasks within a subsystem: a task that generates an *out-const* must be initialized before another task that uses that constant as an *in-const* is initialized.

A two-step initialization and termination is used to support dynamic reconfigurability. High-overhead initialization and termination code is performed during the *init* and *kill* methods respectively, whereas tasks can be activated and deactivated quickly using the *on* and *off* methods. The *on* method is used to update the objects internal state to reflect the current subsystem state. The *off* method is used in cases where the subsystem state must be modified when a task terminates in order to ensure the integrity of the system even after the task stops executing.

The *cycle* method is executed every time the task receives a *wakeup* signal while the task is in the *ON* state. For periodic tasks, the wakeup signal comes from the RTOS timer, whereas for aperiodic tasks, the wakeup signal can result from an incoming message or other asynchronous signaling mechanism supported by the underlying operating system. The *sync* method is used by the aperiodic servers to receive events, and to block if no events are pending.

The Chimera methodology uses a global error handling paradigm to detect and handle faults in the system [37]. An *error signal* is generated whenever an error is encountered. The signal can be caught by either a user-defined or system-defined error handler. By default, an error generated during initialization prevents the creation of the task, and immediately calls the *kill* method which can free any resources that had been allocated before the error occurred. If an error occurs after a task is initialized, then the *error* method is called. The purpose of the *error* method is to either attempt to clear the error, or to perform appropriate alternate handling, such as a graceful degradation or shutdown of the system. If for any reason the task is unable to recover from an error, the task becomes suspended in the *ERROR* state, and a message is sent to the configuration manager indicating that operator intervention is required. After the problem is fixed, the operator sends a *clear* signal, at which time the *clear* method is called. The *clear* method ensures the problem has been corrected. If everything is fine, the task returns to the *OFF* state, and is ready to receive an *on* signal. Otherwise, the error has not been corrected, and the task remains in the *ERROR* state.

The port-based object model allows tasks to be configured based on input constants. Such configuration is performed during a task's *init* method. If one of those constants changes as a configuration changes, the task must be re-initialized, which is accomplished through the *reinit* method. A large bulk of a task's initialization, including creating the task's context and translating symbolic names into pointers, does not have to be re-performed. Only that part of the initialization which is based on the input constants needs to be executed during a re-initialization. The *reinit* method is called automatically when a task that generates an *out-const* is swapped out and a new task generating a *different* value for the same *out-const* is started. The re-initialization ensures that the entire system is dealing with the same constants always, and should a conflict occur, it can be flagged as an error immediately.

With a detailed framework of the port-based object described above, the programmer of a software component only has to define the code for the specific methods. Based on the framework, a C-language specification using abstract data types has been defined and a code template for creating reusable software has been created. The detailed specification and code template are given in [37].

3.5 Reconfigurable Device Drivers

Operating systems provide drivers for input and output devices such as serial ports, parallel ports, and analog/digital converters, but do not make any provisions for the hardware connected to those devices. Such hardware can include sensors such as force, position, tactile, and switches; and actuators, such as robot arms, motors, solenoids, and display devices.

A *reconfigurable device driver* is a port-based object which encapsulates hardware dependencies. It communicates with hardware through the resource ports, and converts raw data into hardware independent information that can be used by generic software modules. Resource ports can be physical devices, such as a serial port, or logical devices, such as a socket. Hardware specific parameters are converted into hardware independent *out-consts*. These are read in by other software modules in the subsystem during initialization, allowing the generic components to configure themselves based on the hardware in use.

As an example, Figure 7 shows a generic Cartesian teleoperation subsystem that can be used with any robot, based on generic forward and inverse kinematics control algorithms [17]. The hardware dependencies of the robot are encapsulated within the *torque-mode robot interface* object. Several modules require *ndof*, the number of degrees of freedom for the robot. In addition, the generic *forward kinematics and Jacobian* and *inverse dynamics* modules require the Denavit-Hartenberg parameters (*dh*) [7], which describes the robot, as input during initialization.

The robot interface module is a reconfigurable device driver, and provides a hardware independent interface to the rest of the subsystem. It generates the *out-consts ndof* and *dh*, shown in Figure 7 as dotted lines. The other modules acquire these parameters during initialization, and configure themselves for the specific robot. If a different robot is used, then only the robot interface module needs to be changed, perhaps generating different *out-consts*. For fixed-configuration robots, the values of *ndof* and *dh* are typically hard-coded within the module or stored in a configuration file, while for reconfigurable robots [32], these values are read in from EPROMs on the robot during initialization of the robot interface module.

3.6 Subsystem Reconfiguration

Intelligent sensor-based control systems are dynamically changing. To use of system resources efficiently, they must be dynamically reconfigurable. For example consider the case of a robotic manipulator which must pick up an object from a table and place it into a final assembly. The robot must first find the object, possibly by communicating with a vision subsystem for tracking the object’s location. Once the object has been located, it moves to a position above the object as quickly as possible, using an appropriate motion control algorithm. The manipulator then uses a force or hybrid control scheme for making contact with the object, then gripper control for grasping the object. Once the object is in hand, the robot again uses a combination of the vision system to search for the target location and motion control to move quickly to that destination. Finally, it reuses the force, hybrid, and gripper control to place the object into the assembly. The application is sequentially decomposed into sub-goals, each of which requires a different subsystem configuration. If the system is not reconfigurable, then all necessary modules must be executing at all times. However, in a dynamically reconfigurable system, only required modules are using the resources at any given time.

The Chimera methodology has been conceived especially to support dynamic reconfiguration. In this section, we demonstrate that capability by use of an example. Figure 8 shows two visual servoing configurations. Both configurations obtain a new desired Cartesian position from a visual servoing subsystem [28], and supply the robot with new reference joint positions. The subsystem configuration in (a) uses standard inverse kinematics, and is used to obtain maximum speed. A similar configuration in (b) uses a damped least squares algorithm, which is slower than using the inverse kinematics algorithm, but it prevents the robot from going through a singularity [46]. The *visual servoing, forward kinematics and Jacobian,* and *position-mode robot interface* modules are the same in both configurations; only the controller module is different.

To support dynamic reconfigurability, either 1) the union of all objects required for the application created, but not necessarily activated, during initialization of the system, or 2) new tasks can be dynamically created in the background prior to becoming activated. As an example, assume that the union of the objects used in Figure 8(a) and (b) are created, and that configuration (a) executes first. The *inverse kinematics* task is turned on immediately after initialization, causing it to run periodically, while the *damped least squares* and *time integrator* tasks remain in the *OFF* state. When the robot is at risk of going through a singularity, a signal is sent to the configuration manager indicating that a dynamic reconfiguration is required. In response, an *off* signal is sent to the *inverse kinematics* task and an *on* signal to the *damped least squares* and *time integrator* tasks. On the next cycle, the new tasks automatically update their own local state variable table, and begin periodic cycling, while the *inverse kinematics* task becomes inactive. Assuming the *on* and *off* operations are fairly low overhead, the dynamic reconfiguration can be performed without any loss of cycles.

For a dynamic reconfiguration which takes longer than a single cycle, the stability of the system becomes a concern. To ensure the integrity of the system, a global *illegal configuration* flag is set when the dynamic reconfiguration begins. It signals to all tasks that a potentially illegal configuration exists. Critical tasks which send signals directly to hardware or external subsystems (e.g. the robot interface module) can go into a locally stable mode. The module then ignores all input variables from other tasks, and instead implements a simple internally-coded local stability feedback loop which maintains the integrity of the hardware. The feedback loop, for example, can keep a robot's position constant or gradually reduce the velocity while the dynamic configuration takes place. Note that the actions to be executed are module dependent and not part of the software framework. When the dynamic reconfiguration is complete, the global flag is reset, and the critical tasks resume taking input from the state variable table.

One issue that arises is determining safe points for performing dynamic reconfiguration. Such an issue must be addressed by the control systems designer, and thus is not addressed by the Chimera methodology. In our systems, we have taken a conservative approach, allowing dynamic reconfiguration to only occur when a robot is at rest. This is implemented by ensuring that a sub-goal is only considered to have been reached once the velocity and acceleration of the robot is zero. Further research into control systems can investigate more aggressive approaches, such as allowing dynamic reconfiguration while the external hardware is still in motion.

The *illegal configuration* flag can also be used when an error is detected in the system. If the execution of one or more modules is halted due to an error, then the state variable data may no longer be valid. To prevent damage to the system, critical tasks go into their locally stable execution until the error is cleared or the system properly shut down. Note that any task with locally stable execution should be considered a critical task for real-time scheduling purposes and thus have highest priority in the system. This ensures that a transient overload during dynamic reconfiguration or error recovery does not preempt the execution of the hard real-time feedback loops in the critical tasks.

The signaling mechanism currently used by Chimera to communicate between the port-based objects and the configuration manager has satisfied our needs, but there is obvious room for improvement. For example, Meta [26], a tool for instrumentizing software components to enable distributed configuration management in a dynamically reconfigurable system, can be used in place of our signaling mechanism. Meta uses a software concept of *sensors* and *actuators* to better monitor and modify configurations to ensure correctness and add robustness during dynamic reconfiguration. The Chimera framework provides an ideal environment for using such a tool, since it is not necessary to instrumentize every software module. Instead, the operating system's framework for the port-based object can be instrumentized, such that all tasks by default use this more advanced configuration signalling tool.

FINAL DRAFT

4. Configuration Verification, Analysis, and Performance

Rapid development of real-time applications is achieved by selecting a set of modules to execute concurrently, allowing the RTOS to handle the timing, synchronization, and communication of the objects. A set of tools that can be incorporated into a configuration manager have been developed to ensure that an application meets its requirements. The tools can be used to verify a subsystem configuration, analyze real-time schedulability on local processors, and analyze performance of the state variable communication and hence the global schedule for a particular configuration.

4.1 Port-Linkage Verification

Input/output port-linkage verification is used to check the correctness of a configuration, such that there is exactly one output port for every input port in the task set, and there are not two modules producing conflicting outputs. In a statically configured application, these checks should be performed by the configuration manager before executing an application. In a dynamic system, the configuration manager should perform these tests before resetting the *illegal configuration flag* to mark a configuration as legal (as discussed in Section 3.6).

The correctness can be 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:

$$(Y_i \cap Y_j) = \varnothing, \text{ for all } i,j \text{ such that } 1 \le i,j \le k \wedge i \ne j \tag{1}$$

and

$$\left(\left(\bigcup_{j=1}^{k} X_j\right) \subseteq \left(\bigcup_{j=1}^{k} Y_j\right)\right) \tag{2}$$

where k is the number of modules in the configuration.

As an example of verifying correctness, consider the configuration that was shown in Figure 4. Assume that module 1 is the *trajectory generator joint position*, module 2 is the *PID joint position controller*, and module 3 is the *torque-mode robot interface*. Therefore $X_1 = \varnothing$. $Y_1 = \{\theta_d, \dot{\theta}_d\}$, $X_2 = \{\theta_d, \dot{\theta}_d, \theta_m, \dot{\theta}_m\}$, $Y_2 = \{\tau_r\}$, $X_3 = \{\tau_r\}$, and $Y_3 = \{\theta_m, \dot{\theta}_m\}$.

From these sets, it is clear that Y_1, Y_2, and Y_3 do not intersect, and hence (1) is satisfied.

To satisfy (2), the union of the input sets and output sets must be taken and compared. We get

$$\cup X = X_1 \cup X_2 \cup X_3 = \{\theta_d, \dot{\theta}_d, \theta_m, \dot{\theta}_m, \tau_r\} \tag{3}$$

and

$$\cup Y = Y_1 \cup Y_2 \cup Y_3 = \{\theta_d, \dot{\theta}_d, \theta_m, \dot{\theta}_m, \tau_r\} \; . \qquad (4)$$

Since $\cup X = \cup Y$, (2) is also satisfied and thus the configuration shown in Figure 4 is legal.

4.2 Real-Time Scheduling Analysis

The foundation of the port-based object is the port-automaton theory, which states that each task executes autonomously and independently of other tasks. This theory provides an ideal model for real-time scheduling analysis. It allows real-time scheduling algorithms such as maximum-urgency-first (MUF)[38] or rate monotonic (RM) [34] to be used without the complexity involved in analyzing task sets with interprocessor communication that force blocking. The MUF algorithm is preferred, since it has a performance improvement over RM, and in addition to guaranteeing execution of hard real-time tasks, it can provide some guarantees for soft real-time tasks [37].

The configuration manager assigns each task to one of the processors in the system. A local real-time scheduling analysis can then be performed for the task sets on each individual processor. Details of scheduling analysis using MUF, including support for hard and soft real-time tasks, aperiodic servers, timing failure detection and handling, and automatic task execution profiling are given in [37]. Although Chimera can dispatch a task to any processor upon request from the configuration manager, it does not define any global scheduling algorithm. Instead, the configuration manager can use any algorithm to map tasks to processors, then use MUF to verify and analyze the task allocation on each processor.

4.3 State Variable Communication Analysis

The global state variable table communication mechanism is a solution for obtaining real-time inter-object communication while maintaining an autonomous execution model for port-based objects in a distributed shared memory environment. The local table allows a task to access required data without contention. The method assumes that the local and global tables are updated every cycle, during the idle times of the tasks. This can only be accomplished if there is sufficient CPU time and bus bandwidth when required. In this section, the effects of communication between the local and global tables on the schedulability of a task set are considered. It is shown that despite constraints and fixed priority hardware, it is possible to provide guaranteed communication and scheduling of critical tasks under some constraints, as discussed later.

First, the time required for transferring data between the local and global tables for each task must be computed. Let t_{IP} be the time required to transfer the *in-vars* and t_{OP} be the time required to transfer the *out-vars* of a port-based object P, assuming no waiting for the bus. These values can be computed as

$$t_{IP} = V_1 + n_I V_a + \sum_{i=1}^{n_{IP}} R(x_{Pi}) \qquad (5)$$

$$t_{OP} = V_1 + n_O V_a + \sum_{i=1}^{n_{OP}} R(y_{Pi}) \qquad (6)$$

where

V_1 = overhead for locking and unlocking the table, excluding waiting time for the bus;

V_a = overhead of transferring each additional variable;

n_{IP}/n_{OP} = number of *in-vars/out-vars* for object *P*;

x_{Pi}/x_{Po} = number of transfers required for the *in-var/out-var i* of object *P*;

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 [15] with a 25MHz MC68030 processor on a VMEbus is shown in Table 1. A VMETRO 25 MHz VBT-321 VMEbus analyzer [45] 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.

Table 1: Breakdown Of VMEbus Transfer Times And Communication Overhead

Operation	Execution Time (µsec)	
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	
lcopy() subroutine call overhead	7	
total overhead for single variable read/write	**34**	$\boldsymbol{V_1}$
additional overhead, per variable, for multivariable copy	**5**	***Va***
raw data transfer over VMEbus, 6 floats	9	***R*(6)**
raw data transfer over VMEbus, 32 floats	31	***R*(32)**
raw data transfer over VMEbus, 256 floats	237	***R*(256)**

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 1 can be substituted into (5) and (6) to estimate the transfer times for each port-based object. As an example, consider the configuration shown in Figure 9, and assume that *ndof=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 2, 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 time the communication of every software module for every type of hardware.

Table 2: 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

For simplicity, (5) and (6) assume 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.

Until now, all measurements and analysis assumed the ideal case where there is no contention for the global table's lock. Next, the worst-case waiting time for the lock by each task is computed.

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:

$$M_j = \max(L_{pj}\Big|_{p=1}^{N_j}) \qquad (7)$$

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 3.3, 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 3.1, 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 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

$$W_k = W_{kLO} + W_{kHI}, \qquad (8)$$

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,

$$W_{kLO} = \max(M_j\big|_{j=k+1}^{r}) \qquad (9)$$

where r is the number of processors.

Next, W_{kHI} is computed. For $k=1$, though, there are no higher priority processors, thus $W_{1HI}=0$ and $W_1=W_{1LO}$. For $k>1$, the potential locking of 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:

$$W_{kHI} = \sum_{j=1}^{k-1} \sum_{i=1}^{N_j} (t_{I,ij} + t_{O,ij}) . \qquad (10)$$

The notation $t_{I,ij}$ is the same as t_{IP}, where object P is referred to by the processor number i and task ID j. For real-time scheduling analysis, the value W_k of each task is added to the worst-case execution time of that task.

As an example, Equations (8), (9), and (10) were applied to the sample configuration shown in Figure 9, with estimated locking times as shown in Table 2. Task periods and cycle times (before adding maximum waiting time) were also arbitrarily assigned to demonstrate the computations, as shown in Table 2. 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 2. The adjusted execution time should then be used for schedulability analysis, as described in Section 4.2.

Table 3: Sample Computations of Worst Execution Time (all times in msec)

Module	Processor	Task ID	Frequency (1/T), in Hz	Period (T)	Worst-Case Execution Time (C), Before Waiting	W_{kLO}	W_{kHI}	W_k	Adjusted worst-case Execution Time (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

In our applications, the average case is significantly lower than the worst case. To compensate, the MUF algorithm's soft real-time capabilities can be used to schedule tasks. MUF soft real-time scheduling makes use of an *effective* execution that can be as low as half the adjusted worst-case execution time in order to improve processor utilization, yet still guarantee that even in the worst case, the tasks will never miss two successive deadlines [37]. Therefore, soft real-time tasks can be placed on the lower priority processors, while the hard real-time tasks should be placed on higher 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 W_{kHI} for tasks on lower priority processors.

A different assignment of tasks to processors can lead to very different results. The configuration manager can use this information as input to compare various configuration possibilities, in order to optimize the global scheduling.

5. User Interface

Program visualization techniques for configuring port-based objects have been developed and incorporated into Onika, a visual programming environment based on the Chimera Methodology in which reusable software can be assembled and executed [11]. This section briefly summarizes Onika.

Onika provides two levels of user interaction:

The *engineer's level* (or lower level) allows control system engineers to create and execute subsystem configurations by assembling port-based objects, as shown in Figure 10. The engineer's level also contains tools for retrieving modules from remote sites, configuring object parameters such as frequency or port names, displaying task execution times (as provided by the Chimera RTOS's automatic profiling [37]) and displaying error messages and signals generated by a port-based object (these features not shown in diagram). A configuration can be saved, and assigned a graphical icon for use with the upper level.

The *application level* (or upper level), allows multiple subsystem configurations to be sequenced together iconically, as shown in Figure 11, in order to create a complete application. The program then executes the configuration corresponding to each icon in order, and dynamically reconfigures the system when moving from one icon to the next. The application level may also contain loops, conditional branching, and parallel execution paths for applications that are decomposed into multiple subsystems (these features not shown in diagram)

Standardized user testing was performed to determine the efficiency of the Onika programming environment. Of the twenty-three subjects tested, all succeeded in learning to use Onika when given a twenty minute tutorial. All subjects then successfully created a simple pick-and-place application involving twenty-four manipulator actions in under seventeen minutes, with the average time being approximately eight minutes. The researchers noticed no major differences in performance with regard to the relative expertise of the subjects in the area of programming; differences resulting in poorer performance typically occurred when either textual descriptions or visual cues were hidden from the subject .

Details of the user testing and more examples of Onika are given in [8] and [9].

6. SUMMARY

This paper describes the Chimera Methodology, a software engineering paradigm that enables the rapid development of real-time applications through use of dynamically reconfigurable and reusable software. It is targeted towards a distributed shared memory computing environment, and implementation has been on a set of single board computer on a VMEbus, hosted by a Sun SPARCstation.

The primary contribution of the Chimera methodology is the port-based object model of a real-time software component. The model is based on applying the port-automaton formal computational theory to object-based design. A finite state machine, detailed interface specifications, and a C-language template are used to define the port-based object. Tools to support the integration, scheduling, and communication between the objects have been developed and incorporated into the Chimera Real-Time Operating System. Techniques for verifying correctness and analyzing performance are also provided for configuration managers that are designed to integrate software created using the Chimera methodology.

The Chimera methodology has been applied successfully to several multi-sensor based applications in labs at Carnegie Mellon University, University of Maryland at College Park, the Air Force Logistics Center, NASA's Jet Propulsion Laboratory, Air Force Institute of Technology, and the National Institute of Standards and Technology. The methodology is also being used by Sandia National Laboratories as a basis for developing virtual laboratories [10].

7. Acknowledgments

The authors would like to thank Matthew W. Gertz for providing the diagrams in Section 5 and offering a graphical interface for demonstrating the use of port-based objects. They would like to thank Wayne Carriker for creating *modmaker*, a tool which automatically generates the C code template for a specific object given the port interface of the module. They would also like to thank the other members of the Advanced Manipulators Laboratory at Carnegie Mellon University for creating the initial library of reusable software components for robotics based on the Chimera Methodology.

8. References

[1] B Abbott et al, "Model-based software synthesis," *IEEE Software*, pp. 42-52, May 1993.

[2] M. A. Arbib and H. Ehrig, "Linking Schemas and Module Specifications for Distributed Systems," in *Proc. of 2nd IEEE Workshop on Future Trends of Distributed Computing Systems*, Cairo, Egypt, September 1990.

[3] B. W. Beach, "Connecting software components with declarative glue," in *Proc. of International Conference on Software Engineering*, Melbourne, Australia, pp. 11-15, May 1992.

[4] T. E. Bihari, P. Gopinath, "Object-oriented real-time systems: concepts and examples," *IEEE Computer*, vol. 25, no. 12, pp. 25-32, December 1992.

[5] W. Blokland and J. Sztipanovits, "Knowledge-based approach to reconfigurable control systems," in *Proc. of 1988 American Control Conference*, Atlanta, Georgia, pp. 1623-1628, June 1988.

[6] G. Booch, "Object-oriented development," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 211-221, February 1986.

[7] J. J. Craig, *Introduction to Robotics*, 2nd Ed., (Reading, Massachusetts: Addison Wesley), 1989.

[8] M. W. Gertz, *A Visual Programming Environment for Real-Time Sensor-Based Systems*, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, November 1994. Available on-line, URL=*http://www.ee.umd.edu/~dstewart/publications/thesis/gertz94.ps.Z.*

[9] M. W. Gertz, R. A. Maxion, and P. K. Khosla, "Visual Programming and Hypermedia Implementation within a Distributed Laboratory Environment," to appear in *Journal of Intelligent Automation and Soft Computing*, 1995.

[10] M.W. Gertz, D.B. Stewart, B. Nelson, and P.K. Khosla, "Using hypermedia and reconfigurable software assembly to support virtual laboratories and factories," in *Proc. of 5th International Symposium on Robotics and Manufacturing (ISRAM)*, Maui, Hawaii, pp. 493-500, August 1994.

[11] M.W. Gertz and P.K. Khosla, "Visual programming and program visualization techniques for port-based objects," in Proc. of *1994 Annual Meeting of the American Nuclear Society*, New Orleans, LA, June 1994.

[12] A. Gheith, B. Mukherjee, D. Silva, and K. Schwan, "KTK: Kernel support for configurable objects and invocations," in *Proc. of Second Int'l Workshop on Configurable Distributed Systems*, Pittsburgh, PA, pp. 92-105, March 1994.

[13] M. Gorlick, "A visual platform for the synthesis of complex systems", in *Proc. of the 1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop* (CSESAW '94), Silver Spring, MD, pp. 233-244, July 1994.

[14] N. Haberman, D. Notkin, "Gandalf: software development environments," *IEEE Transactions on Software Engineering*, vol.12, no.12, pp. 1117-1127, Dec. 1986.

[15] Ironics Incorporated, *IV3230 VMEbus Single Board Computer and MultiProcessing Engine User's Manual,* Technical Support Group, 798 Cascadilla Street, Ithaca, New York 14850.

[16] R.K. Jullig, "Applying formal software synthesis," *IEEE Software*, vol.10, no.3, pp. 11-22, May 1993.

[17] L. Kelmar and P. K. Khosla, "Automatic generation of forward and inverse kinematics for a reconfigurable modular manipulators systems," in *Journal of Robotics Systems*, vol.7, no.4, pp. 599-619, August 1990.

[18] J. Kramer, J. Magee, and K. Ng, "Graphical configuration programming," *IEEE Computer*, vol. 22, no. 10, pp. 53-65, October 1989.

[19] G. K. Krikor, M. T. Raza, and D. B. Stewart, "Design of a real-time co-operating system for multiprocessor workstations," in *Proc. of Hawaii International Conference on System Sciences, Software Technology Track*, pp. 499-507, Jan. 1996.

[20] D. A. Lamb, "IDL: Sharing intermediate representations," ACM Transactions on Programming Languages and Systems, vol.9, no.3, pp. 297-318, July 1987.

[21] D. M. Lyons and M. A. Arbib, "A formal model of computation for sensory-based robotics," *IEEE Transactions on Robotics and Automation*, vol 5, no. 3, pp. 280-293, June 1989.

[22] J. Magee, N. Dulay, and J. Kramer, "Structuring parallel and distributed programs," *IEE Software Engineering Journal*, vol. 8, no. 2, pp. 73-82, March 1993.

[23] J. Magee, J. Kramer, and M. Sloman, "Constructing distributed systems in Conic,", *IEEE Transactions on Software Engineering*, vol. SE-15, no. 6, Dec. 1989.

[24] J. Magee, J. Kramer, M. Sloman, and N. Dulay, "Configuring Object-Based Distributed Programs in REX," *IEE Software Engineering Journal*, vol. 8, no. 2, pp. 139-149, March 1992.

[25] B. Markiewicz, "Analysis of the computed-torque drive method and comparison with the conventional position servo for a computer-controlled manipulator," Technical Memorandum 33-601, The Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, March 1973.

[26] K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman, "Tools for distributed application management," in *IEEE Computer*, vol. 24, no. 8, pp. 42-51, August 1991.

[27] L. D. Molesky, C. Shen, G. Zlokapa, "Predictable synchronization mechanisms for multiprocessor real-time systems," *The Journal of Real-Time Systems*, vol. 2, no. 3, September 1990.

[28] B. Nelson and P.K. Khosla, "Integrating Sensor Placement and Visual Tracking Strategies," in *Proc. of Experimental Robotics III: The Third International Symposium*, Kyoto, Japan, October 1993.

[29] 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.

[30] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. of 10th Int'l Conf. on Distributed Computing Systems*, Paris, France, pp. 116-123, June 1990.

[31] C. Rattray, J. McInnes, A. Reeves, and M. Thomas, "Knowledge-based software production: from specification to program," in *UT IK 88 Conference Publication*, pp. 99-102, July 1988.

[32] D. E. Schmitz, P. K. Khosla, and T. Kanade, "The CMU reconfigurable modular manipulator system," in *Proc. of the International Symposium and Exposition on Robots* (designated 19th ISIR), Sydney, Australia, pp. 473-488, November 1988.

[33] K. Schwan, P. Gopinath, and W. Bo, "Chaos: kernel support for objects in the real-time domain," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 904-916, August 1987.

[34] L. Sha and J. B. Goodenough, "Real-time scheduling theory and Ada," *IEEE Computer*, vol.23, no.4, pp. 53-62, April 1990.

[35] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," Carnegie Mellon University, CS Dept., Tech Report #CMU-CS-87-181, November 1987.

[36] T. E. Smith and D. E. Setliff, "Towards an automatic synthesis system for real-time software," in *Proc. of Real-Time Systems Symposium*, San Antonio, Texas, pp. 34-42, December 1991.

[37] D. B. Stewart, *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems*, Ph.D. Dissertation, Carnegie Mellon University (Pittsburgh, PA) April 1994. Available on-line, URL=*http://www.ee.umd.edu/~dstewart/publications/thesis/dstewart94.ps.Z*; 196 pages.

[38] D. B. Stewart and P. K. Khosla, "Real-time scheduling of sensor-based control systems," in *Real-Time Programming*, ed. by W. Halang and K. Ramamritham, (Tarrytown, New York: Pergamon Press Inc.), 1992

[39] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "The Chimera II real-time operating system for advanced sensor-based robotic applications," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 6, pp. 1282-1295, November/December 1992.

[40] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Integration of real-time software modules for reconfigurable sensor-based control systems," in *Proc. 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '92)*, Raleigh, NC, pp. 325-333, July 1992.

[41] M. Steenstrup, M. A. Arbib, and E. G. Manes, "Port automata and the algebra of concurrent processes," *Journal of Computer and System Sciences*, vol. 27, no. 1, pp. 29-50, August 1983.

[42] A. K. Tetewsky, "GUI development for real-time applications", *Dr. Dobb's Journal*, vol.19, no.6; pp. 28-41, June 1994.,

[43] S. Vestal, "Mode Changes in a Real-Time Architecture Description Language" in *Proc. of Second Int'l Workshop on Configurable Distributed Systems*, Pittsburgh, PA, pp. 136-146, March 1994.

[44] R. J. Vetter, C. Spell, and C. Ward, "Mosaic and the World-Wide Web," *IEEE Computer*, vol. 27, no. 10, pp. 49-57, October 1994.

[45] VMETRO Inc., *VBT-321 Advanced VMEbus Tracer User's Manual*, 2500 Wilcrest, Suite 530, Houston, Texas 77042.

[46] C. Wampler and L. Leifer, "Applications of damped least-squares methods to resolved-rate and resolved-acceleration control of manipulators," *ASME Journal of Dynamic Systems, Measurement, and Control*, vol. 110, pp. 31-38, 1988.

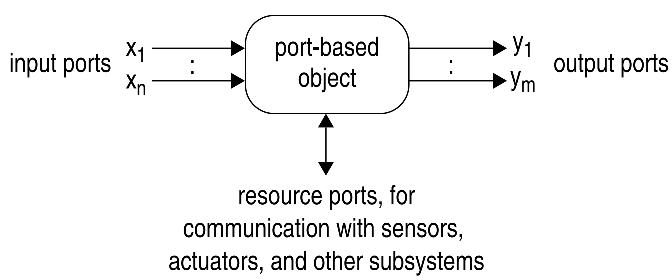


Figure 1: Simplified model of a port-based object

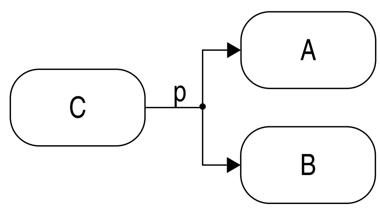


Figure 2: Fanning an output into multiple inputs

..

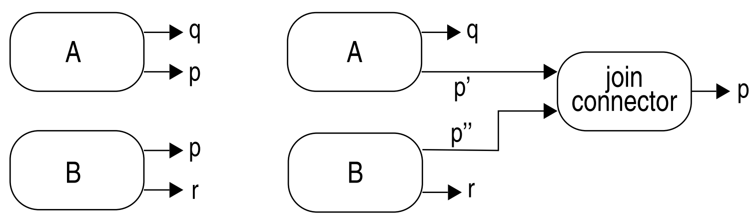


Figure 3: Joining multiple outputs into a single input

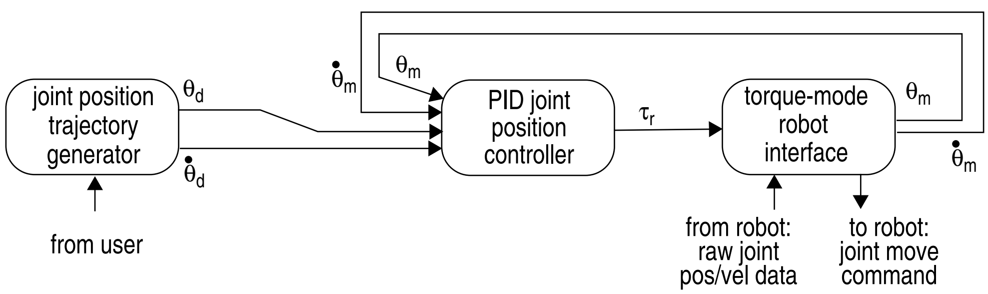


Figure 4: Example of PID joint control.

..

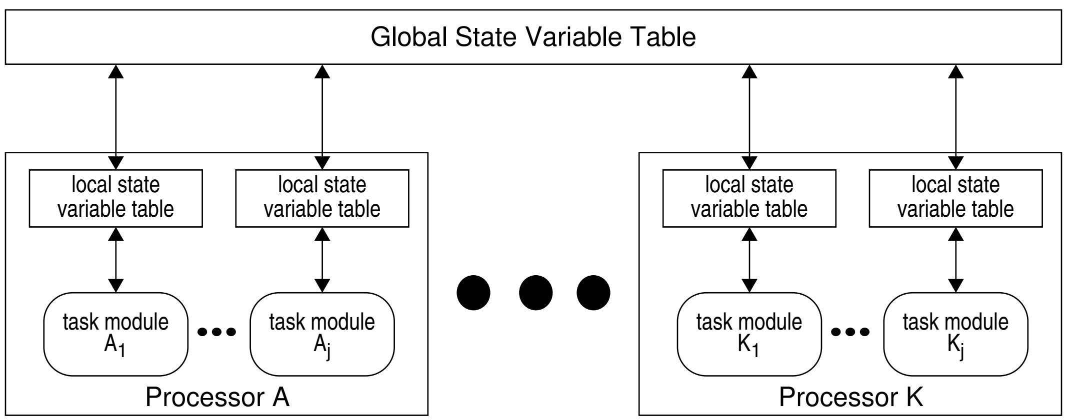


Figure 5: Structure of state variable table mechanism for port-based object integration

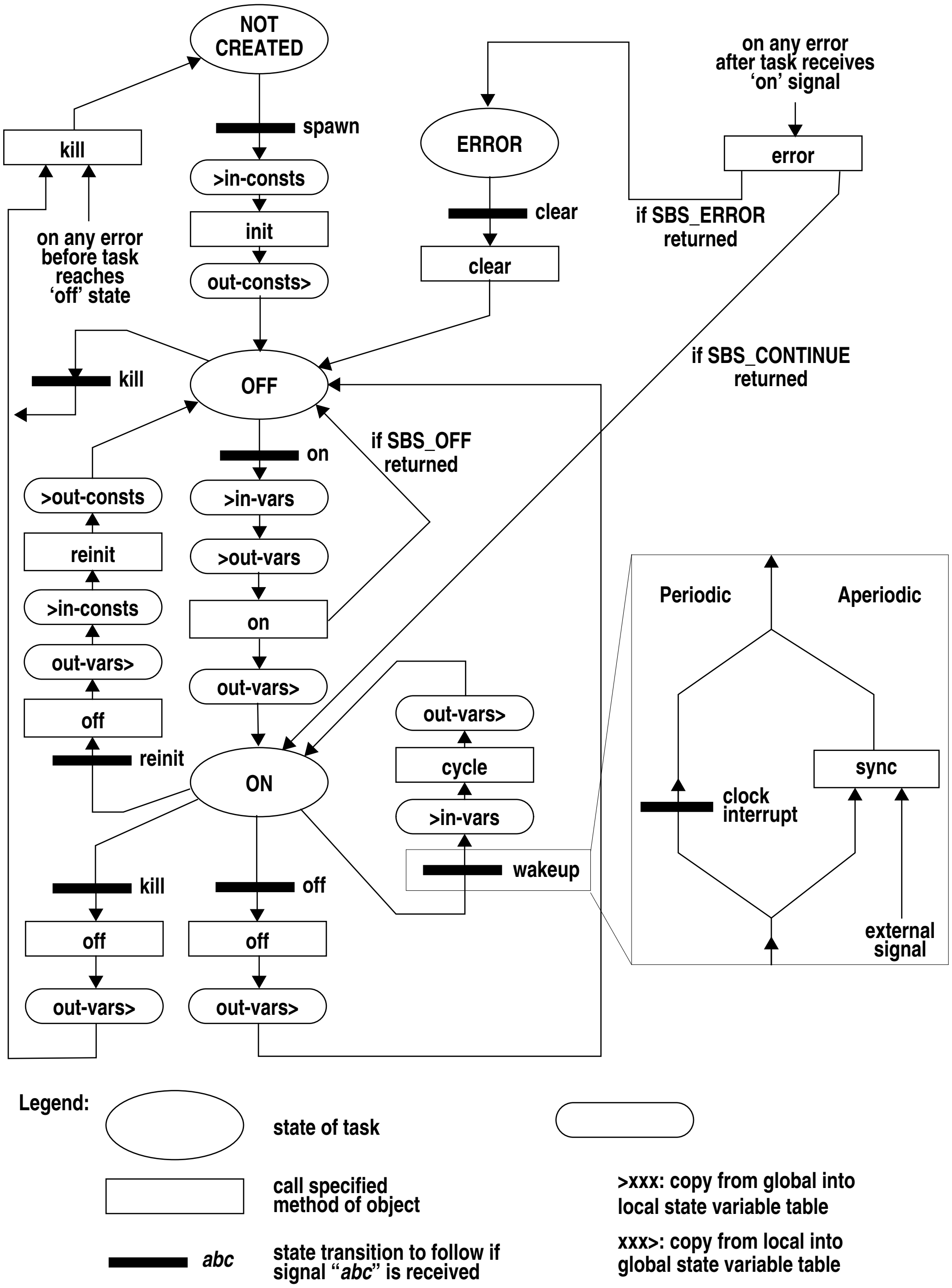


Figure 6: Finite state machine model of a port-based object, with state transtions shown as process-flow diagrams.

.

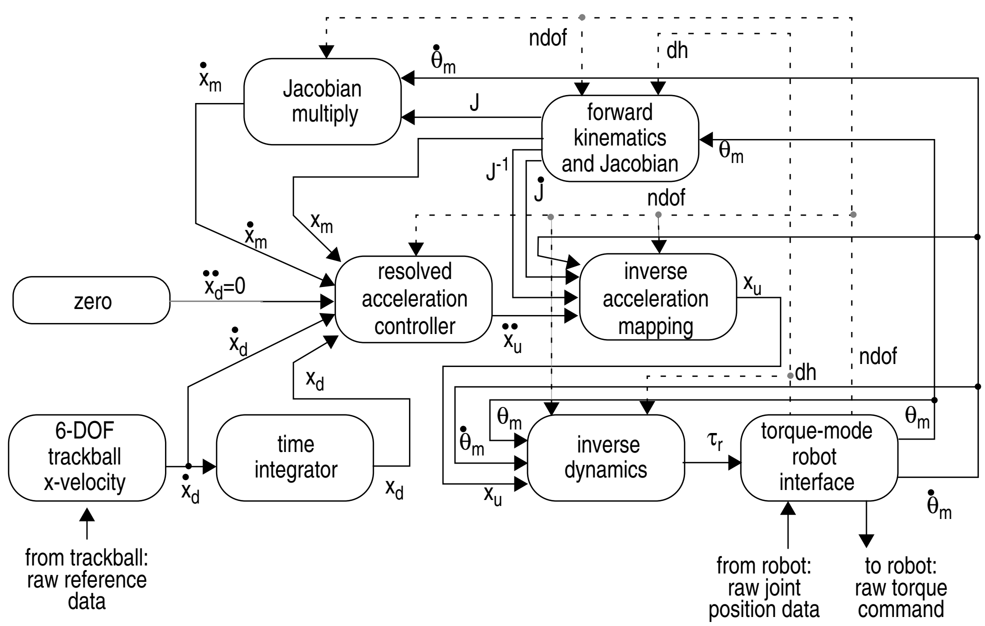
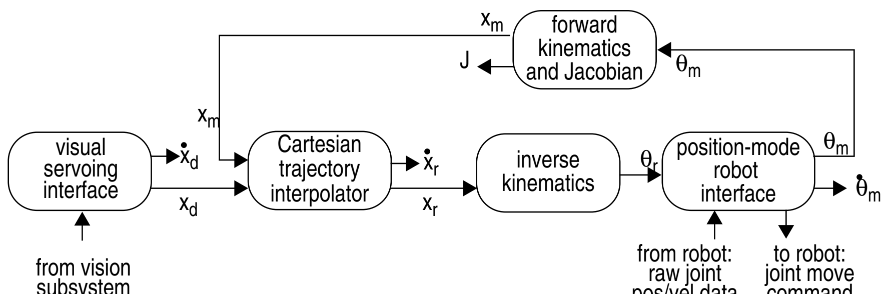
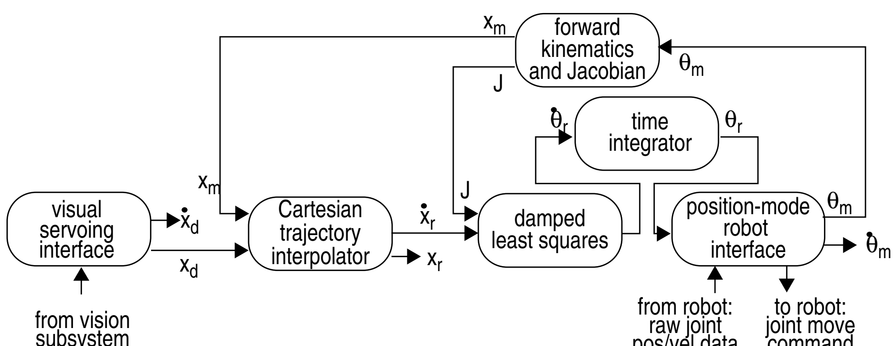


Figure 7: Example of module integration: Cartesian teleoperation

.



(a)



(b)

Figure 8: Example of dynamic reconfiguration of a visual servoing application: (a) using inverse dynamics module; (b) using damped least squares module

.

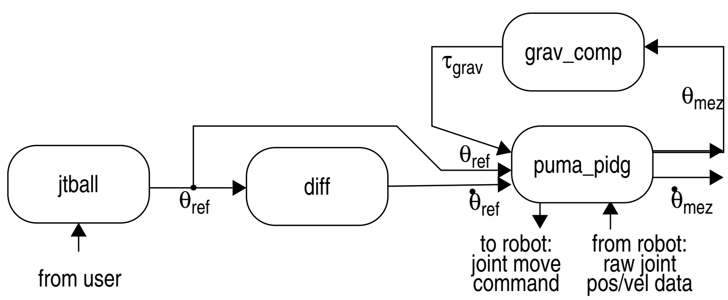


Figure 9: Configuration for sample analysis of state variable communication.

.

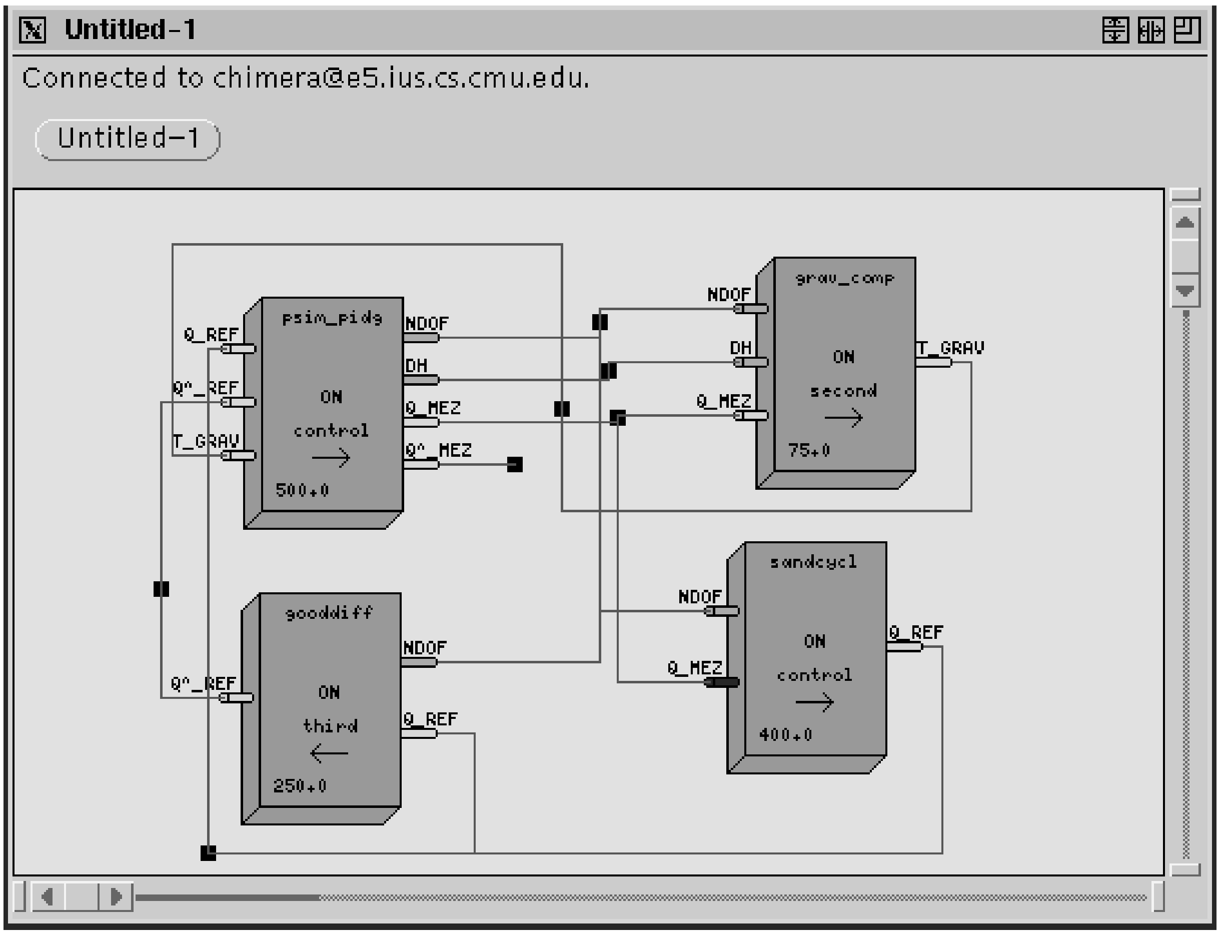


Figure 10: Example of Onika's engineer's level for graphically integrating port-based objects.

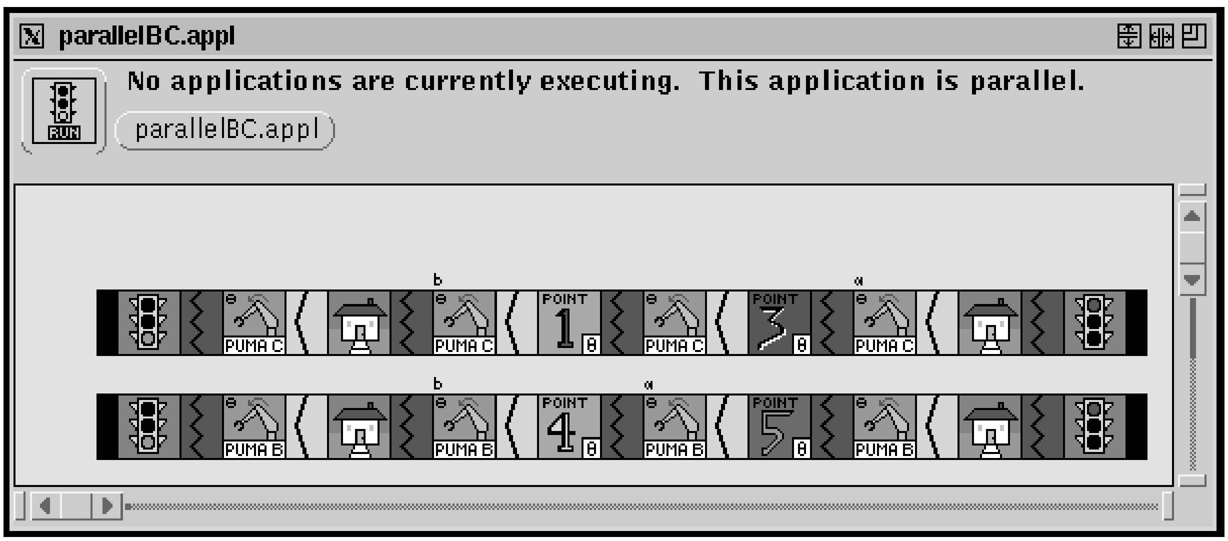


Figure 11: Example of Onika's application level for creating applications based on subsystem configurations of port-based objects.