

## The Chimera Methodology: Designing Dynamically Reconfigurable Real-Time Software using Port-Based Objects

David B. Stewart\* and P. K. Khosla†

\*Dept. of Electrical Engineering and Institute for Advanced Computer Studies  
University of Maryland, College Park, MD 20742

†Dept. of Electrical and Computer Engineering and The Robotics Institute  
Carnegie Mellon University, Pittsburgh, PA 15213

### Abstract

*The Chimera Methodology is a new software engineering paradigm which addresses the problem of developing dynamically reconfigurable and reusable real-time software. The foundation of the Chimera methodology is the port-based object model of a reusable software component. The model is obtained by applying the port-automaton formal computational model to object-based design. Global state variable table real-time communication is used to integrate port-based objects, which eliminates the need for writing and debugging glue code. The Chimera real-time operating system provides tools to support the software models defined by the Chimera methodology, so that real-time software can be executed predictably using common real-time scheduling algorithms. A hypermedia user interface has been designed to allow users to easily assemble the real-time software components that are designed based on the Chimera methodology. Use of the methodology can result in a significant decrease the development time and cost of real-time applications.*

### 1: Introduction

The Chimera Methodology is a new software engineering paradigm for designing and implementing real-time software for multi-sensor systems. The Chimera name was used since this methodology was a direct result of our work on the Chimera real-time operating system project [28].

The methodology addresses the problem of developing dynamically reconfigurable component-based real-time software. Transfer and reuse of real-time software is difficult and often seemingly impossible due to the incompatibility between hardware and systems software at different sites. This has meant that new technology developed at one site must be reinvented at other sites, if in fact it can be incorporated at all. Technology transfer, therefore, has been a very expensive endeavor, and the reuse of software from previous applications has been virtually non-existent.

The use of component-based software has been proposed to improve software development time by addressing the software reuse and technology transfer issues [26]. For example, a user developing a real-time application may need a specific software algorithm developed at some other site. Currently,

The research presented in this paper is supported, in part, by Sandia National Laboratories, NASA, and the author's institutions. Partial funding for D. Stewart was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a graduate fellowship.

users may go to the library or search through the network for keywords, then find a book or journal article describing the mathematical theory or computer science algorithm. After they have printed a copy of the paper, they read the article closely, then spend significant time writing, testing, and debugging code to implement the algorithm. Once that is done, they write more code to integrate the new algorithm into their existing system, and perform further testing and debugging. This process can easily take days or weeks of the person's time for each algorithm needed for their application, and thus take many months to complete the programming of an entire application.

The ultimate application programming environment should allow for complete software reuse and the ability to quickly transfer technology from remote sites. Users who need a specific algorithm would search through a global distributed software library based on the hypermedia information system currently available with the World-Wide Web [32]. When they find a suitable book or article, they not only get the written theory, but they can also follow a link to a reusable software module created by the authors. The algorithm would already be programmed, fully tested, and debugged. With an action as simple as a mouse-click, that software algorithm is copied into the user's personal library, and is ready to be used in their application. This process would take a few minutes at most. The user could then create their application by putting together these software building-blocks through the use of a graphical user interface. Within hours, a complete application could be assembled, as compared to the months that it would take to do so using conventional methods.

This process of assembling an application without writing or automatically generating any new glue code is called *software assembly* [26]. In this paper, we describe the Chimera methodology, which defines new software models, communication protocols, and interfaces for supporting software assembly.

### 2: Related work

There has been significant research in the area of software reuse, with three major directions emerging: software synthesis, interface adaptation, and object design.

Software synthesis, also known as automatic code generation, generally employs artificial intelligence techniques such as knowledge bases [1] [3] [24] and expert systems [12] [20] to generate the "glue" code for automatically integrating reus-

able modules. As input, they receive information about the software modules, the interface specifications and the target application, and as output produce code using both formal computation and heuristics. For a truly generic framework, however, it is desirable that the integration of software be based on the interfaces alone, and not on the semantics of the modules or application, as the latter results in the need for large knowledge bases. Furthermore, software synthesis only allows for statically configuring an application, and usually does not support dynamic reconfiguration.

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 [11] [14] [16]. 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 [4] [19]. None of these methods have been adapted to real-time systems, and there are no clear extensions which would ensure that interface adaptation and communication between modules can be performed in real-time.

The Chimera methodology differs from interface adaptation and software synthesis methods of integrating software, in that it addresses the actual design of software components, rather than just addressing their interfaces.

Object design is a popular software modelling technique which can form the basis for software reuse. An object is a software entity which encapsulates data and provides methods as the only access to that data. Wegner distinguishes between two types of object design methodologies: *object-based design* (OBD) and *object-oriented design* (OOD) [7]. Whereas OBD only defines the encapsulation of data and access to that data, OOD is an extension which also defines the interrelation and interaction between objects. The interrelation of objects in OOD is defined through inheritance using the notions of classes, superclasses, and meta-classes [34]. 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. 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 [5]. For example, The Chaos Real-Time Operating System [22] was designed to use object-oriented design with 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 [5], 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.

The work presented in this paper is an alternate approach which avoids the real-time problems associated with object-oriented design, while maintaining the advantages of using objects to obtain modular encapsulation, a necessary basis for software reusability. It combines object-based design with the port-automaton computational model, as described next, to model dynamically reconfigurable real-time software components.

Streenstrup and Arbib [30] 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 [15], 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. The next section presents the details of the model.

### 3: Port-Based Objects

A new abstraction for real-time software components that applies the port automaton theory to object-based design is the port-based object. 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 dia-

grams, we represent such fanning of the output with just a dot at the intersection between two 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's output is based on 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 user or software assembly tool (such as Onika [9]) can modify the output port names of modules with the same outputs using the aliasing features provided by the RTOS, 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.

### 3.1: Object Integration using State Variables

A task set is formed by linking multiple objects together to form either an open-loop or closed-loop subsystem. Each

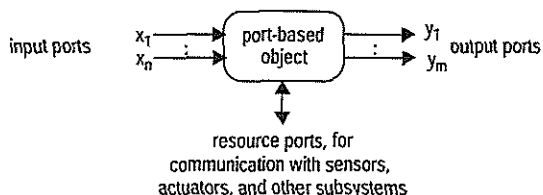


Figure 1: Simplified model of a port-based object

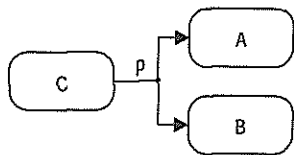


Figure 2: Fanning an output into multiple inputs

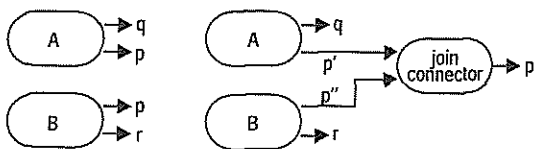


Figure 3: Joining multiple outputs into a single input

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 every 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 terms, the analysis and implementation complexity of real-time scheduling algorithms such as *maximum-urgency-first* [27] and *rate monotonic* [23] is minimized.

The port-automaton model assumes that the most recent data is always present at the input ports, which implies that the ports are *not* message queues. 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 time to output data a function of the number of external tasks. This contradicts the automaton model where an object is not aware of its external environment.

State variables provide an alternative to messages. A subsystem state can be 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. This method guarantees that the most recent data is always available as a state variable. However, this also introduces a problem of integrity.

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 blocking terms to the real-time scheduling analysis and create the possibility of priority inversion and deadlocks. We now present our solu-

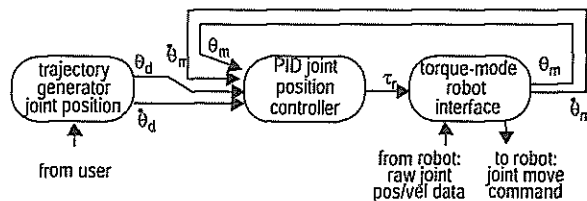


Figure 4: Example of PID joint control.



tions for obtaining the required synchronization while maintaining an autonomous execution model.

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 may argue that locking the CPU may lead 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  $\mu$ sec in order to perform a system call such as a full context switch [31]. 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.

In most sensor-based control applications, the volume of data is small. For example, for 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 5  $\mu$ sec on a CPU with a 100  $\mu$ sec context switch time, considering that a context switch may contain as many as 200 operations for saving and restoring registers and updating a process control table.

One notable exception in which the small volume of data 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 one of the resource ports [25]. For a vision subsystem, configurable inter-object communication can be implemented using high-volume data streams and synchronized tasks [10], instead of states and asynchronous tasks as described in this paper. Synchronous systems are much 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 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 for applications such as active vision [18].

For multiprocessor environments, we have designed a *global state variable table mechanism* for the inter-object communication [29], since locking only one of the CPUs will not provide the necessary atomic execution, and locking all the CPUs is not feasible. The mechanism is based on the combined use of global shared memory and local memory for the exchange of data between modules, as shown in Figure 5. The global state variable table is stored in the shared memory. The variables in this table are a union of the input port and output port variables of all the modules that can be configured into the system. 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. Details of the global state variable table mechanism are given in [29].

### 3.2: Configuration Verification

In order to ensure that the data required by a port-based object is always available, a configuration analysis is required.

A legal configuration exists when there is exactly one output port for every input port in the task set, and there are no two modules which produce the same output. The correctness of a configuration can be verified analytically using set equations, where the elements of the sets are the state variables. A configuration is legal only if

$$(Y_i \cap Y_j) = \emptyset, \text{ for all } i, j \text{ such that } 1 \leq i, j \leq k \wedge i \neq j \quad (1)$$

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

where  $X_j$  is a set representing the input variables of module  $j$ ,  $Y_j$  is a set representing the output variables of module  $j$ , and  $k$  is the number of modules in the configuration.

As an example, consider the configuration 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  $X_1 = \emptyset$ ,  $Y_1 = \{\theta_d, \dot{\theta}_d\}$ ,  $X_2 = \{\theta_d, \dot{\theta}_d, \theta_m, \dot{\theta}_m\}$ ,  $Y_2 = \{\tau_r\}$ ,  $X_3 = \{\tau_r\}$ , and  $Y_3 = \{\theta_m, \dot{\theta}_m\}$ .

From these sets we can easily see 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

$$\bigcup X = X_1 \cup X_2 \cup X_3 = \{\theta_d, \dot{\theta}_d, \theta_m, \dot{\theta}_m, \tau_r\} \quad (3)$$

and

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

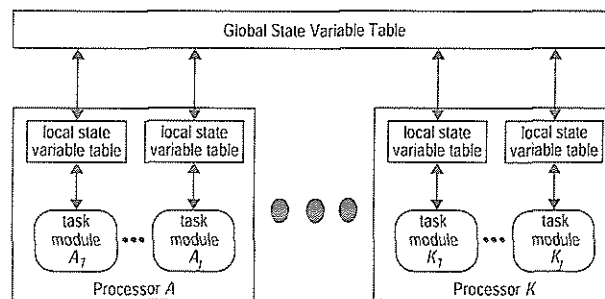


Figure 5: Structure of state variable table mechanism for port-based object integration

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

A task set consisting of port-based objects provides a good model for real-time scheduling analysis, because each task executes autonomously and independently of other tasks. Such characteristics are desirable, as they allow for real-time scheduling using algorithms such as maximum-urgency-first [27] or rate monotonic [23], without the complexity involved in task sets with inter-task dependencies which can result in significant blocking, priority-inversion, or deadlock. The maximum-urgency-first algorithm is preferred, since it has both a performance improvement over rate monotonic, and it accounts for the fact that not all tasks have to be hard real-time. For example, many sensor tasks are soft real-time, in that they can tolerate the occasional missed deadline as long as the object can extrapolate the data to account for the missed cycle.

Details of the maximum-urgency-first real-time scheduling algorithm used by the Chimera RTOS, including support for hard and soft real-time tasks, aperiodic servers, timing failure detection and handling, and automatic task execution profiling are given in [25].

### 3.3: Detailed Port-Based Object Model

In order to ensure autonomous execution of a port-based object, the local and global state variable tables must be updated regularly, in such a way that the updates never occur while a task is executing. To address this issue, we first discuss a new programming paradigm for implementing software, then provide a detailed model of a port-based object in support of that paradigm.

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, synchronization, and integration with other modules.

Chimera uses an "inside-out" method of programming as compared to the traditional methods of developing software. Rather than the software modules invoking the RTOS whenever an operating system service is required, the RTOS services are always executing, and they invoke 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 new programming paradigm is also highly desirable because the operating system is in total control of every task, and as a result enables automatic execution time profiling for tasks, and allows the operating system to detect timing failures and handle them accordingly [25].

In order to demonstrate the principles of the new programming paradigm, a detailed diagram of the port-based object model is presented. Every port-based object in a configuration

is a real-time task on one of the processors, and has the structure shown in Figure 7.

The ellipses show the possible states of the task, which can be *NOT-CREATED*, *OFF*, *ON*, or *ERROR*. A task that is in the *OFF* state has been created, meaning that a context for the task exists, but that the task is in a suspended state waiting for a 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 task. The *ERROR* state is for tasks that have encountered unrecoverable errors during their execution.

The transition between states occurs as a result of signals sent by a subsystem interface to the RTOS. The subsystem interface signals can come from a user through a command-line or graphical interface, from an external subsystem, or from another task in the same subsystem in the case of an embedded system. These signals are shown in Figure 7 as solid bars.

The methods of an object are invoked by the RTOS in response to the signals from the subsystem interface, and form part of the state transitions. The methods are shown as rectangular boxes. Each object has each of the following special methods: *init*, *on*, *cycle*, *off*, *kill*, *error*, *clear*, *reinit* and *sync*. The *init* and *on* components are for a two-step initialization. The *cycle* component executes once for each cycle of a periodic task, or once for each event to be processed by an aperiodic server. The *off* and *kill* components are for a two-step termination. The *error* and *clear* components are for automatic and manual recovery from errors respectively. The *reinit* component is used for dynamic reconfiguration. The *sync* component is used by aperiodic servers to receive events through a port-based object's resource ports. More details on each of these methods is given below.

Before and after each method of a port-based object is executed, a transfer is made between the local and global 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 7 as oval boxes.

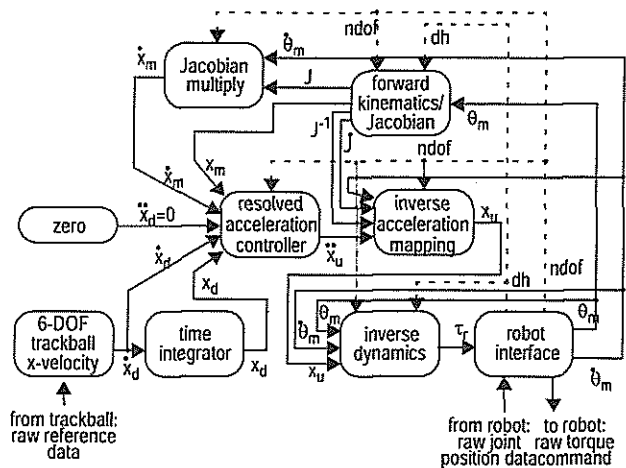


Figure 6: Example of module integration: Cartesian teleoperation

# Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

## Real-Time Litigation Alerts



Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

## Advanced Docket Research



With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

## Analytics At Your Fingertips



Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

## API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

## LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

## FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

## E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.