To appear in Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '95), Pittsburgh, PA, August 1995.

Rapid Development of Robotic Applications using Component-Based Real-Time Software

David B. Stewart* and P. K. Khosla†

*Dept. of Electrical Engineering and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742; Email: *dstewart@eng.umd.edu*

†Dept. of Electrical and Computer Engineering and The Robotics Institute
Carnegie Mellon University, Pittsburgh, PA 15213; Email: *pkk@cmu.edu*

***Abstract**: Component-based real-time software speeds development and lowers cost of robotics applications. It enables the use of rapid prototyping or incremental software process models. The Chimera Methodology is a software engineering paradigm targeted at developing and integrating dynamically reconfigurable and reusable real-time software components. It is founded upon the notion of port-based objects. The focus of this paper is how to apply the Chimera Methodology specifically to the development of robotic applications.*

1. Introduction

Component-based real-time software speeds development and lowers cost of robotics applications. It enables the use of rapid prototyping or incremental software process models.

The Chimera Methodology is a software engineering paradigm targeted at developing and integrating real-time software components [8]. It is founded upon the notion of port-based objects. It combines the port automaton computation model of a concurrent process with object-based design of software. The result is a software model for objects that can be reused, statically integrated and dynamically reconfigured, supports real-time applications, and can be used in a distributed shared memory environment.

The focus of this paper is how the component-based software methodology can be applied specifically to robotic application development. The focus is not on the Chimera Methodology itself.

Relevant details of the Chimera Methodology are summarized in Section 2. Our classification of software components and terminology are given in Section 3. Rapid software development using component-based technology is described primarily through use of examples in Section 4. The real-time operating system requirements and current limitations of the methods presented in this paper are discussed in Section 5. Finally, the research is summarized in Section 6.

2. Background

The *port-based object* is an abstraction for real-time software components that applies the port automaton computational model for concurrent processes [1][11] to object-based design [13]. 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. Input and output ports can contain either *constant* or *variable* data. Constant data is used during initialization for configuring generic software. Variable data is updated every cycle. A port-based object is shown diagrammatically in Figure 1

A task set is formed by linking multiple objects together to form either an open-loop or closed-loop subsystem. Port-names are used to perform the binding between objects. Each object in the subsystem executes as a separate task on one of the processors in a distributed environment. An example of a fairly simple closed-loop subsystem is the PID joint control of a robot, as shown in Figure 2. 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. To support this model, a task obtains the most recent data available from its input ports at the beginning of each cycle. At the end of the cycle, after perform-

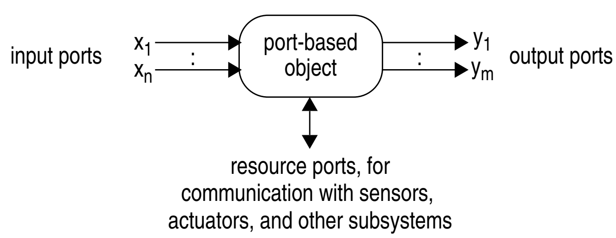

Figure 1: Diagrammatic model of a port-based object

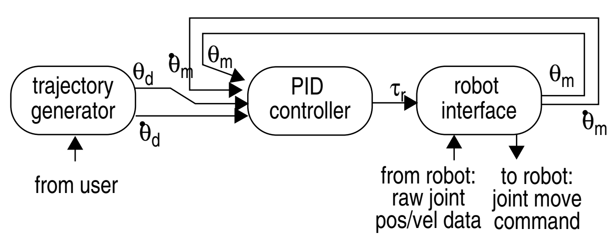

Figure 2: Example of a PID joint control using port-based objects.

The research presented in this paper is supported, in part, by Sandia National Laboratories, NASA, and the author's institutions.

ing any necessary computations, it places new data onto its output ports. The task is 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 interfacing hardware.

In order to support the port-automaton model, communication between tasks is performed through a global state variable table mechanism, as shown in Figure 3. 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 local and global tables are updated to always contain the most recent data, and the local table is never updated while a task is using the table [9].

The Chimera Methodology is targeted towards distributed shared memory hardware architectures, in order to support real-time applications. Only Chimera is specific for dynamically reconfigurable real-time systems implemented in a distributed shared memory environment. An in-depth comparison of the Chimera Methodology to other frameworks for component-based software is presented in [8]. For sake of brevity, it is not repeated here.

3. Reusable Software Components

A software component is a module or object which performs a specific function according to a set of specifications.

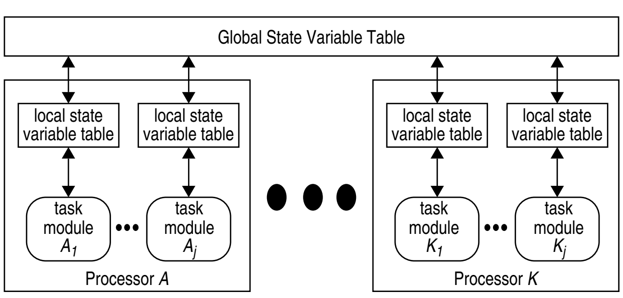

Figure 3: Structure of state variable table mechanism for port-based object integration

Its interface should be well defined, so that the component can easily be integrated with other components. A software component can have internal state. An external state can be shared with another software component only if the shared state is itself an entity, such that each module can import or export any part of it. Multiple software components can execute either sequentially or concurrently. In a concurrent system, appropriate synchronization is required to ensure the integrity of any shared state. The Chimera model of a port-based object satisfies all these requirements.

There are four types of software components: generic, hardware dependent, application dependent, and integration. These types are described next.

3.1 Generic Components

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

An example of a generic component for robotics is a module that performs the forward kinematics using the Denavit-Hartenberg (DH) parameters [2]. The DH parameters are used to describe the link and joint characteristics of a robotic manipulator. A forward kinematics generic component can be configured for any manipulator by obtaining these parameters during initialization. It is the responsibility of hardware dependent components to provide configuration information such as the DH parameters.

3.2 Hardware Dependent Components

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

An HD interface component is used to convert hardware dependent signals into hardware independent states, such that other generic components can make use of the information. The functionality of these components is similar to a device driver. A device driver is used by a real-time operating system (RTOS) to provide hardware-independent access to I/O devices, such as serial ports, parallel ports, analog/digital converters, and frame grabbers. The HD interface components create a hardware independent interface to the hardware that is connected to these I/O devices, such as robotic actuators, switches, sensors, and displays.

An example of an interface component is a robot interface software module. Different robots require different signals to control them. For example, a PUMA robot can be controlled through a Trident Robotics Robot Control Board [12], while a Robotics Research Arm is controlled through a Multibus-based custom controller. In either case, however, these modules convert data read from the robot's sensors into the current measured position and velocity of each joint, and convert the desired position or torque of each joint into control signals to the robot's actuators. Although the software component is hardware dependent, it provides a hardware independent interface so that it can be integrated with generic components.

An HD computational component does not communicate directly to hardware. Rather, it provides similar functionality as a generic component, but typically with better performance. For example, the generic forward kinematics module described above can be used with both the PUMA and Robotics Research manipulators. It is also possible to create two other software modules, one called the *PUMA forward kinematics*, and the other called the *Robotics Research forward kinematics*. If the interface of these HD computational components is the same as the *generic forward kinematics* module, then the generic and HD components are interchangeable. Such interchangeability is an essential ingredient for rapid prototyping, improving system performance, and fault tolerance, as discussed further in Section 4.2.

3.3 Application Dependent Components

Application dependent (AD) components are modules used to implement the specific details of an application. As the name implies, these components are not reusable across different applications. Ideally. AD components are not required, since they must be redeveloped for each new application.

An example of an AD component is a module that instructs a robot to follow a specific trajectory.

Modules initially defined as an AD component can usually be transformed into a generic component by converting hard-coded information into variable input. The input can be obtained from the user, a configuration file, or another module.

3.4 Integration Components

Integration components are used to integrate the generic, hardware dependent, and application dependent components. These components are often known as the *glue* code for an application. Typical integration components include communication mechanisms, schedulers, and user interfaces. The communication mechanisms are used to transfer data between other software components. The schedulers are used to control access to system resources. The command interfaces are used to provide hooks into an application which allow a user to modify application parameters or allow the application to cooperate with other applications.

Integration components require interaction with all other components in the system. As a result, these are often the most time-consuming modules to create.

Much of the research in developing the Chimera Methodology addresses these integration components. The solution that forms part of the methodology is to create a standard set of integration components, and to incorporate them as services provided by the underlying RTOS [8]. Therefore, the need to write new integration components is eliminated and development time and cost for new applications is significantly reduced. See Section 5 for a discussion of the RTOS services required to support the Chimera Methodology.

4. Rapid Development of Robotic Applications

Robotic applications can be developed rapidly by assembling reusable software components. In this section, rapid development is demonstrated through use of examples.

The development of an application can be performed in three stages:

1. Development of reusable software components;
2. Assembling components to create a task set;
3. Sequencing task sets to create an application

These stages are now demonstrated through use of examples.

4.1 Stage 1: Development of Software Components

The first stage is to develop the software components. Once a component is developed, it is placed into a library, and does not have to be redeveloped again. Therefore, as the software libraries become more complete, the need for creating new components is reduced. Ideally, an application can be developed without the need for developing any new components, in which case this stage can be skipped.

To create these modules, the programmer simply fills in the blanks of a C-language template. The template code handles all of the object's interfacing, communication, and synchronization. The programmer must only add the algorithms to produce the output response based on the input response. The template is defined in [10].

In preparation for the examples given to demonstrate the next stage, suppose that the generic components shown in Table 1 and the HD components shown in Table 2 are developed, and placed into a robotic manipulator domain library. Assume that initially there are no application dependent components required.

Table 1: Example of software components in a hardware-dependent software library

Name	Function	Description
gfwdkin	Generalized Forward Kinematics	Compute Forward Kinematics based on the DH-parameters obtained during initialization of the module.
ginvkin	Generalized Inverse Kinematics	Compute Inverse Kinematics based on the DH-parameters obtained during initialization of the module.
cinterp	Cartesian Trajectory Interpolator	Given the current measured position and the desired final position, compute the intermediate reference positions

4.2 Stage 2: Assembly of Reusable Components

The second stage is to create a task set by assembling reusable components [7]. In this section, several examples of task sets are given to demonstrate the use of the different types of software components, and how they can all be interchanged when defined using the port-based object model. The first example is Cartesian teleoperation of a Reconfigurable Modular Manipulator System (RMMS). In the second example, the RMMS is replaced with a Puma 560. In the third example, the performance of the Puma 560 is improved. In the fourth exam-

Table 2: Example of software components in a hardware-dependent software library

Name	Function	Description
rmms	RMMS Interface	Hardware dependent interface to the Reconfigurable Modular Manipulator System (RMMS)
puma	Puma 560 Interface	Hardware dependent interface to Puma robot, using the Trident Robotics TRC004 [12] to bypass VAL
pfwdkin	Puma Forward Kinematics	Compute Forward Kinematics for a Puma 560 Robot.
pinvkin	Puma Inverse Kinematics	Compute Inverse Kinematics for a Puma 560 Robot.
tball	6-DOF Trackball Interface	A hardware dependent interface to a 6-degree-of-freedom trackball.

ple, an application dependent component is created to convert the teleoperated system into an autonomous system.

The assembly of software components at this stage can be performed either by a a command line interface, a graphical user interface (GUI), or for autonomous assembly from a C-program. Software assembly using a command line or C-program is described in detail in [9]. The easiest method, however, is to use the GUI. To select a module is as easy as clicking on an icon representing a component in a library. The GUI then shows not only the modules selected, but all the port connections as well. An example of a task set displayed by *Onika* [3], a GUI designed especially to support port-based objects, is shown in Figure 4.

Cartesian teleoperation of an RMMS

The RMMS is a robot which can quickly be constructed out of standard joints and links [6] . As such, it is the most flexible robotic manipulator available. Due to its generality, the configuration of the robot is not known until run-time, and thus it is not feasible to use hardware specific control algorithms. Rather, control algorithms such as the forward and inverse kinematics must be generalized, typically based on the DH parameters [5] .

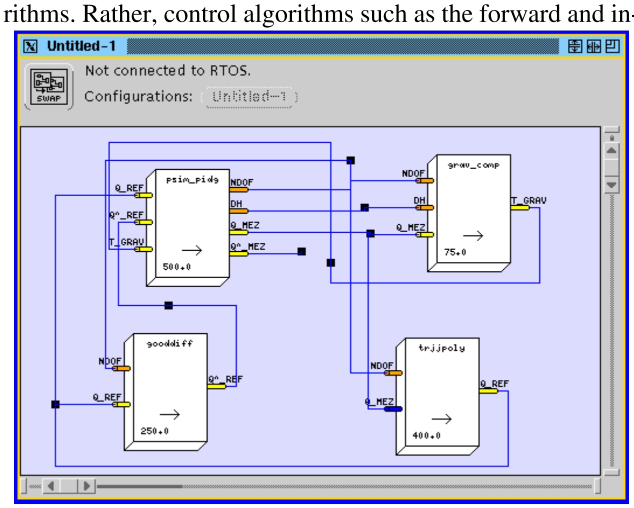

Figure 4: Example of Onika's engineer's level for graphically assembling port-based objects.

A task set for Cartesian teleoperation of the RMMS can be created by using the *tball*, *cinterp*, *gfwdkin*, *ginvkin*, and *rmms* components, as shown in Figure 5. The dotted lines represent configuration constants, while the solid lines represent state variables. Both the constants and variables are transferred between software components through the SVAR mechanism.

During initialization, the *rmms* module reads the EPROMs of the robotic manipulator, and determines the n_{dof} and the *DH* parameters for the particular configuration of the RMMS. Those parameters are then output as configuration constants.

The *gfwdkin* and *ginvkin* modules obtain the configuration constants as input, and thus configure themselves for the robot in use. Task frequency is also a configuration parameter for all tasks. After initialization, each task executes at its pre-determined frequency, and the task set can be scheduled using any popular real-time scheduling algorithm [9].

The *tball* and *cinterp* modules do not require any configuration constants from the RMMS, since it outputs data as Cartesian coordinates, which is independent of the robot in use.

Cartesian teleoperation of a Puma 560

Suppose that a Puma 560 robot is to be used instead of the RMMS. The *rmms* module can be replaced with the *puma* robot interface module, as shown in Figure 6.

Since the Puma 560 is a fixed configuration robot, its n_{dof} and *DH* parameters are constant. Instead of reading these values from the robot, they can instead be hard-coded into the *puma* module, and output as configuration constants. There is no need to change any other module, since the *gfwdkin* and

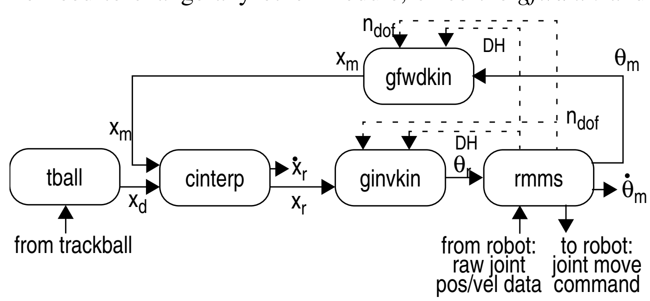

Figure 5: Example of Cartesian teleoperation of an RMMS using generic software components.

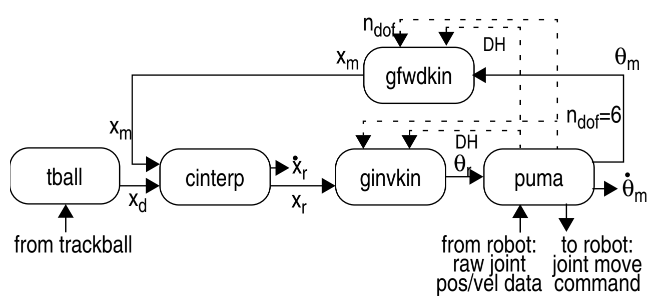

Figure 6: Example of Cartesian teleoperation of a Puma 560 using generic software components.

ginvkin modules will configure themselves during initialization for the Puma 560 based on the new values of n_{dof} and *DH*.

The reconfigurable device driver concept enables vendors of robots and sensors to provide software to quickly integrate their hardware into existing applications. If their software and the target application are compliant with the same port-based object interface, then replacing either the RMMS or Puma 560 with a different manufacturer's manipulator is simply a matter of changing the one robot interface module. For this reason, HD interface components are also referred to as *reconfigurable device drivers*. Consequently, applications using new robotic hardware can quickly be created by assembling a reconfigurable device driver with existing generic components.

Improving performance of a Puma 560

The primary drawback of generic modules is that they may not be efficient. For example, the computation of the forward kinematics based on the DH configuration constants and using matrix operations will naturally be slower than performing similar computations for a specific robot, such as the Puma 560, where the DH parameters are constant, and unnecessary computations (such as multiply by zero or 1) can be eliminated.

In order to improve the performance of an application, an HD computation component can be created. The *pfwdkin* and *pinvkin* modules are examples of such components. They compute the forward and inverse kinematics respectively for a Puma 560, and are faster than their generic counterparts. It is then desirable to replace *gfwdkin* with *pfwdkin*, and *ginvkin* with *pinvkin*, as shown in Figure 7.

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

Even when an HD computational component is created to replace a generic component, it does not eliminate the usefulness of the generic component. In order to improve fault tolerance of an application, the generic component can still be used as a standby module, or it can execute in parallel with the HD computational component, albeit at a lower frequency, in order to provide consistency checks. Further research in this area has begun, with the goal of rapidly developing fault tolerant applications.

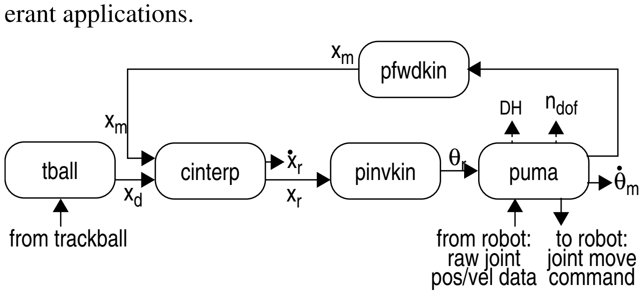

Figure 7: Example of Cartesian teleoperation of a Puma 560 using hardware dependent computational components.

Autonomous Execution of a Puma 560

As an example of an AD component, suppose that a custom trajectory module *ctraj* is created to replace the teleoperation. The *ctraj* module can replace the *tball* module for controlling the RMMS as shown in Figure 8.

This example shows how an AD component can be integrated into the system by defining it as a port-based object. Furthermore, even though a module is application dependent, it does not have to be hardware dependent. Thus if the hardware for the application is changed, the AD component does not have to be change. Figure 9 demonstrates this by replacing *rmms* with *puma*, but not having to change *ctraj*.

4.3 Stage 3: Application Development

The third stage is to sequence task sets to create a subsystem, where an application contains one or more subsystems executing in parallel. Development at this level uses the dynamic reconfiguration capabilities of the Chimera implementation of port-based objects [8].

Every function to be performed by the manipulator can be developed independently, by designing a task set for that one function only. That configuration is then saved as an entity, called a *job*. Jobs for robotic applications can be simple functions, such as *move to point X* or *pick-up object*, or can be more complex multi-sensor feedback loops, such as *visually track destination and move to it* or *explore shape of object using tactile sensing*. As shown in Section 4.2, separate configurations can also be created for switching between autonomous and teleoperated control or to use different hardware.

A subsystem is developed by sequencing jobs in order to achieve the desired goal. This step is performed using a GUI,

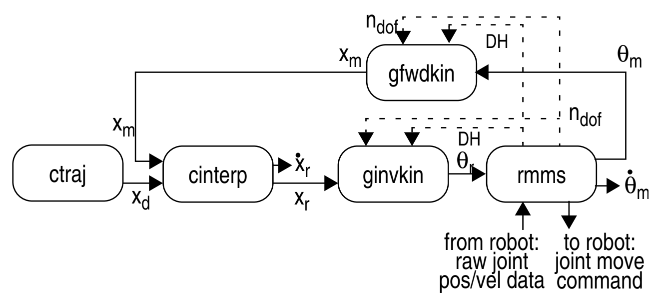

Figure 8: Example of autonomous execution of an RMMS using generic software components.

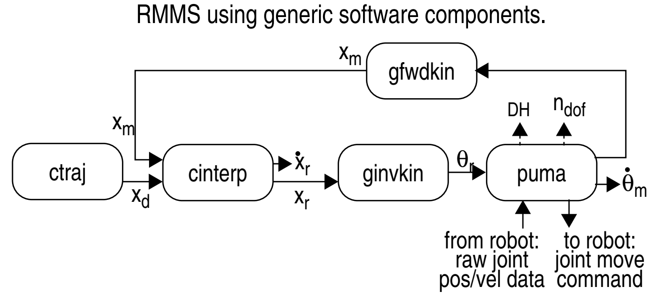

Figure 9: Example of autonomous execution of a Puma 560 using hardware dependent computational components.

where a job is represented by an icon, as demonstrated in Figure 10.

Each job icon represents a complete task set that can be analyzed, tested, and timed independently. When the application is run, the task set corresponding to the first job is initialized and run. When that job completes, the subsystem dynamically reconfigures such that tasks not needed are deactivated, and new tasks are activated. Tasks that are common to both task sets (such as the robot interface module, which would be common to all jobs in a subsystem) continue to execute.

The Chimera implementation of the task manager and global state variable table implementation allow tasks to be activated and deactivated in real-time [9]. The primary concern is maintaining stability of the hardware during reconfiguration. Currently, this is achieved by using a pessimistic approach: dynamic reconfiguration is only performed when the physical hardware — i.e. the robot —is at rest. Using a more agressive approach, dynamic reconfiguration can be performed while the robot is still in motion, as long as input to the robot interface module remains continuous through the transition from one job to the other. Research into such more aggressive approaches is continuing.

5. Discussion

The Chimera Methodology is described in sufficient detail for readers who want to replicate the model for their own use in [8][9]. These documents define precise structures and interfaces for reusable software components, and describes the operating system services that can be incorporated into a multiprocessor RTOS to support port-based objects. The minimal services required include a distributed task manager, a global state variable table communication mechanism, and global error handling. Optional services that improve the environment include an advanced real-time scheduler which supports aperiodic servers and soft real-time tasks, automatic task profiling, configuration file utilities, and a graphical user interface. These services have all been implemented as part of the Chimera RTOS, and is described in detail in [10].

Error handling is performed using a global error handling paradigm. Whenever an error is encountered, an error signal is generated. Any task, but typically the task manager, can catch the error, and perform handling as necessary. If the task manager catches the error, it may dynamically reconfigure the subsystem to execute a pre-defined error handling task set.

The Chimera Methodology described in this paper makes two basic assumptions: 1) the task set is to be executed in a distributed shared memory environment; and 2) volume of data transferred between port-based objects is small. Both these assumptions are required for predictable communication

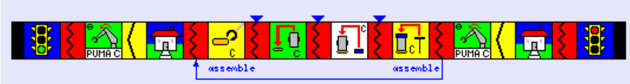

Figure 10: Example of Onika's application level for creating applications by sequencing jobs.

using the global state variable table. We are investigating alternate communication mechanisms which would allow us to relax both of these assumptions, and to use the Chimera Methodology in other real-time applications.

6. Summary

This paper describes the development of robotic applications using component-based real-time software, based on the Chimera Methodology. The combined use of generic, hardware dependent, and application dependent hardware provide an ideal environment for rapid prototyping and incremental software process models.

7. References

[1] 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.

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

[3] M. W. Gertz, *A Visual Programming Environment for Real-Time Sensor-Based Systems*, Ph.D. Dissertation, Dept. of Electrical and Computer Engr, Carnegie Mellon University, Nov. 1994; *http://www.glue.umd.edu/~dstewart/bib/thesis/.*

[4] 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.

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

[6] 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.

[7] D. B. Stewart, M. W. Gertz, and P. K. Khosla, "Software assembly for real-time applications based on a distributed shared memory model," in *Proc. of the 1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop (CSESAW '94)*, Silver Spring, MD, pp. 217-224, July 1994.

[8] D.B. Stewart, P. Khosla, "The Chimera Methodology: Designing Dynamically Reconfigurable and Reusable Real-Time Software using Port-Based Objects," to appear in International Journal of Software Engineering and Knowledge Engineering, Sept. 1995.

[9] 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. *http://www.glue.umd.edu/~dstewart/bib/thesis/.*

[10] D. B. Stewart and P. K. Khosla, *Chimera 3.0 Real-Time Programming Environment*, Program Documentation, Dept. of Electrical and Computer Engr, Carnegie Mellon University, Pittsburgh, PA 15213; *http://www.glue.umd.edu/~dstewart/bib/manuals/.*

[11] 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.

[12] Trident Robotics and Research, Inc., "TRC004 User's Manual," 2516 Matterhorn Dr., Wexford, PA 15090.

[13] P. Wegner, "Concepts and paradigms of object-oriented programming," *OOPS Messenger*, vol.1, no.1, pp.7-87, August 1990.