APHRODITE: Intelligent Planning, Control and Sensing in a Distributed Robotic System

Aleksandar Timcenko Steven Abrams Peter K. Allen*
Center for Research in Intelligent Systems
Department of Computer Science
Columbia University
New York, NY 10027

Abstract: In this paper we describe a general-purpose robot programming environment APHRODITE[1] built at Columbia University's Center for Research in Intelligent Systems. The environment is based on a distributed multiprocessor architecture that offers great flexibility and computing strength, and provides a reliable real-time response, while using off-the-shelf software tools. The supporting software that we developed supports transparent operation across the network and distributed interupt handling, We define the interface between main hierarchical levels in the system and describe the implementation as it has been used to create a distributed, multi-tasking application. We have implemented a test task that consists of assembly operation that includes compliant motion planning in an uncertain environment and the intelligent monitoring of the assembly process.

1. Introduction

A typical robotic system is usually comprised of many different functional processing modules. One of the major problems impeding progress in robotics is the integration of these modules in a coherent way. These modules usually span a variety of different hardware, data structures, and communication paths with widely varying bandwidth. In addition, there is a need to express concurrency and synchronization within these systems comprised of interacting parts.

The development of robot programming environments has been recognized to be lagging behind the advances in hardware architectures and programming environments in general [6]. Many complex robot control systems employ multiprocessing; supporting software environments need address the issues of interprocessor communications and software abstractions to fully utilize the potential of multiprocessor systems in robotics, yet relatively few robot programming systems do this in a coherent fashion. While there are numerous results in

*This work was supported in part by DARPA contract N00039-84-C-0165, NSF grants DMC-86-05065, DCI-86-08845, CCR-86-12709, IRI-86-57151, IRI-88-1319, CISE grant CDA-90-24735, North American Philips Laboratories, Siemens Corporation and Rockwell International.

[1] Yes, we have come up with a long acronym to name our project: Autonomous Programmable Hierarchical Robot Object-oriented Distributed Intelligent Task Executor (APHRODITE). We sincerely hope that it will obtain its rightful place in the pantheon of big acronyms.

increasing the raw computational power of robot controllers through employing parallelism in control law evaluation, there is relatively little research and implementation work done in analyzing multitasking controller structures.

The level of sophistication required by modern robot systems equals, if not surpasses, the functionality that only the most developed programming environments can meet. The most demanding among these requirements are:

- graphical user interface and "comfortable" programming environment
- modular, extensible architecture
- reliable real-time responsiveness
- transparent distributed execution

On the other hand, the diversity of robotic subsystems has grown rapidly over the course of years imposing more and more complex and computationally demanding tasks on the controller's processor. The possibilities that multiprocessing offers were recognized early and a plethora of multiprocessor-based robot controllers have been investigated and developed. The advantages gained by multiprocessing are the raw computing power of the overall system on one hand, but this is counterbalanced by the difficulties of maintaining and programming complex distributed systems. The analyses of software support for distributed, concurrent controllers has drawn relatively less attention than the analysis of the computational benefits of parallelism. This is due, in part to the lack of a "standard" hardware platform and underlying operating system for distributed robot control.

Nevertheless, there have been many important advances in this area. The idea of embodying a robot controller in a multitasking environment is not new [2, 9]. Since the early 1980s, it has been steadily pursued, resulting in implementations of very complex multi-robot multi-CPU systems. One example of such a system is given in [3]. Another very complex robotic system (described in [12]) utilizes a special-purpose concurrent programming environment that combines vision, motion planning and control functions. Results presented in [16] demonstrate differences in real-time characteristics which depend on the software structure of interprocess communication.

The problems that developers of complex software systems for robotic applications usually face were recognized in [7]. Some of them are the difficulty (or, quite frequently, inability) to reuse and extend the existing software, and hence the high cost of software development. Another problem is the management of a system's growing complexity. Their proposed solution is important for the design of distributed controllers since it employs an object-oriented abstraction of robotic subsystems and attempts to define the services each of those subsystems is capable of providing (or, in the object-oriented language, messages it is capable of receiving).

Distributed robot controllers consist, in one way or another, of three main components: the programming environment, the real-time processing and the communication channel between those two. Not all of these three components are equally developed in all approaches; by including more functionality in one, another may be left rudimentary. For example, RCCL [4, 5] augments an existing general-purpose operating system (UNIX) to support real-time functions. However, RCCL supplies just the rudimentary real-time kernel to the robot controller, whose job is to merely convey the joint coordinates to per-joint processors. On the other side of the spectrum are so-called real-time operating systems (examples of which include CHAOS [11], VxWorks [17], CHIMERA [10] and VMEexec [8]) that provide full-feature operating system's kernels with provisions for strict timing, scheduling and interrupt handling, required by real-time applications.

The third component, the communication channel, hasn't received much attention, which is partially due to the simple underlying hardware it is usually based on — dual-ported memory or a bus window, that does not require complex supporting software. Nevertheless, there are certain disadvantages inherent in that solution, such as limited extensibility, special-purpose hardware required (dual-ported RAM, bus-to-bus adaptors) and lack of a standardized interface.

The diversity in software and hardware solutions for these three components usually results in integration that is heavily application-dependent and essentially has to be reimplemented from scratch for every new application, as well as the tendency to use slower communication devices (such as serial lines) merely for the sake of ease of programming, deliberately sacraficing more powerful mechanisms (such as networks) that may be more difficult to use. Also, there are features that are implemented only rudimentarily in the existing network communication software. Here we have in mind especially methods for asynchronous communications (interrupts).

This paper describes a framework we have developed to address the problem of integrating a distributed robot system, with a focus on three critical aspects of robotic tasks: planning, control and sensing. The ability to integrate easily all three of these aspects allows a user to build a sophisticated and capable robotic system. Moreover, because the system is able to "glue" modules together in a coherent way, research and refinement can be done on a module by module basis without affecting the overall system structure or capability. Thus, new algorithms and strategies can be easily integrated into already existing systems, allowing for rapid testing and improvement of system modules. The goal of this work was to build an extensible system based on standard, "of-the-shelf" UNIX programming tools that unifies disparate robotic tasks under one programming paradigm and provide a testbed for a distributed, network-based environment for programming very complex robotic tasks. We have demonstrated the system by programming a moderately complex robotic assembly task that has as its core a number of research modules that have been developed independently as stand-alone processes.

2. System Software Design

There are several goals we tried to achieve in the design of the APHRODITE software:

- provide transparent access to distributed robot devices and processes (similarly to the X windowing system)
- define the methods for handling asynchronous events over the network (i.e. distributed interrupts)
- define unified methods for including new devices into the system
- shield application programmer from the details of the distributed implementation
- use of-the-shelf software tools rather rewriting major operating system features

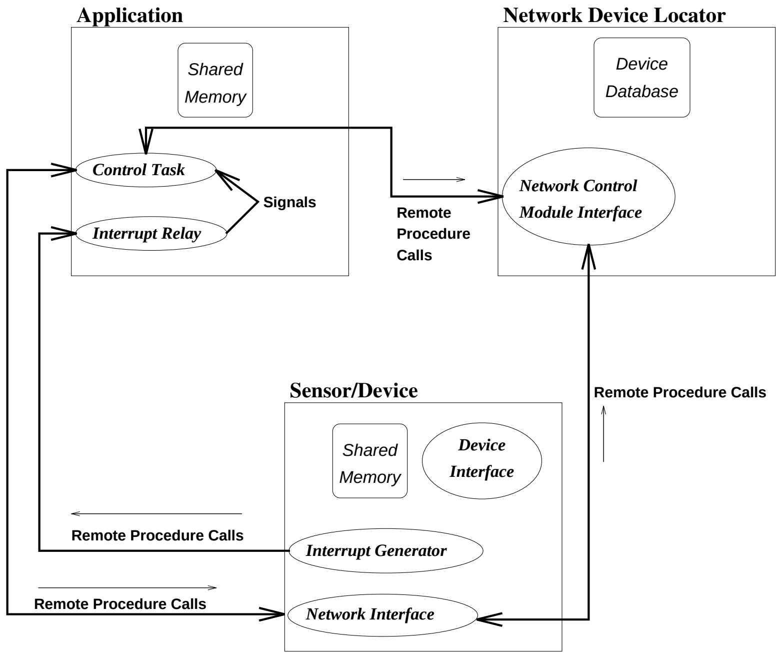

Figure 1: Architectural Overview: Boldface arrows indicate data flow. thin arrows indicate direction of procedure call. Ovals indicate processes. Boxes group logical modules.

The APHRODITE system achieves the above goals using standard UNIX features such as signal handling, shared memory and remote procedure calls (RPCs) (see figure 1).

APHRODITE views a robot system as being comprised of an application and a collection of planning and sensing modules, each of which is referred to as a *device* (even if it is simply a computational planning task and not a physical device). In addition, APHRODITE provides a *Network Device Locator* (NDL) which keeps track of the locations and status of the available devices in the system, relieving the application of this burden. In fact, the only globally known data in the environment is the RPC program number of the NDL itself. Clients use RPC broadcasts to contact the nearest NDL and obtain a *device handle* for communicating with the requested devices.

(The NDL can be replicated on the network for robustness.) From APHRODITE's point of view, each *device* consists of up to three processes:

Device Interface (DI): Maintains an up-to-date view of the device (i.e. the current sensor readings or the current results of the planning task) in a shared memory segment. This may require interfacing to a physical device via a serial line, dual-ported memory, or other hardware connection.

Network Interface (NI): Fields requests from the application and communicates with the NDL. Accesses device data through shared memory.

Interrupt Generator (IG): Responsible for generating *distributed interrupt* signals to the client application, as required.

An APHRODITE application consists of two processes:

Control Task (CT): The application, from the user's point of view. Communicates with devices via high-level abstractions which shield the programmer from the architecture of the underlying hardware.

Interrupt Relay (IR): Signals the control task upon the appearance of a *distributed interrupt.*

In the following subsections, we describe how the application, devices, and NDL interact with each other during the startup and execution of an application.

2.1. Device wakeup procedure

Before any activity can take place, the devices must be awakened or "booted" (this is normally part of each system's startup sequence). As a device server starts up, the DI establishes the hardware connection to the physical device. It invokes the NI process which notifies the NDL (via a broadcast RPC) about the device's existence. NI then registers itself on the local host as an RPC server and listens for incoming RPCs from client applications. DI and NI communicate through a piece of shared memory.

After hearing from a device's NI, the NDL updates its local database of active devices on the network.

2.2. Opening a device

When an application needs to access a particular sensing or planning device, it issues a device-open request. First, the application broadcasts an open request to the NDL to find the host address and program number of the device's NI. Next, the application issues an RPC to the device's NI to request either exclusive or shared access to the device.

2.3. Distributed interrupts

Distributed interrupt handlers are a unique feature of APHRODITE. They allow the application to be interrupted via a standard UNIX signal when a predefined event is detected by a device, even if the application and the device's support software are running on different computers. The application invokes the IR task, whose job it is to signal the CT in the event of an interrupt.[2] IR registers itself on the local host as an RPC server and gives CT its program number. CT then makes an *interrupt install* RPC to the device's NI giving it an *interrupt mask* and the program number of the IR task.

When the NI receives the *interrupt install* request, it creates an IG (if none has been created yet). Using the interrupt mask, the device knows under what conditions it should interrupt the application. IG monitors the shared memory to see if the interrupt conditions are met. When these conditions are met, the device will make an RPC to the application's IR, sending it whatever data is related to the interrupt. Then, the IR stores the data in shared memory and sends a standard UNIX signal to the CT indicating what event caused the interrupt. The CT's standard UNIX interrupt handler can access the data in shared memory and act accordingly.

2.4. Normal data flow

During the course of the task, the DI insures that the device's shared memory always has a current view of the device's state. The application CT queries and commands devices via RPC's, and the device's NI responds with data and status. Meanwhile the device's IG monitors shared memory for interrupt conditions and calls the application's IR when appropriate.

[2] In UNIX, signals work across the same architectural boundaries as `fork()`. Specifically, in a multiprocessor, a process created by `fork()` may run on a different processor than its parent, but that is transparent to the signaling mechanism.

2.5. *Summary*

The important aspects of the software system are:

- The application need not have any a priori information about where distributed devices are located.
- The Device Interface is free to communicate with the device as rapidly as possible, while the Network Interface fields client requests asynchronously. This insures that the client gets the most up-to-date information possible without disturbing the Device Interface and without flooding the network with device polls.
- The device can asynchronously interrupt the application at predefined events, (i.e. guarded moves, impending collision, robot joint velocity exceeded, etc.).

It is important to remember that the term *device* has been used for both physical devices such as sensors and robots as well as higher-level tasks such as trajectory planners. Using this paradigm, it is possible (and, in fact, desirable) to build hierarchies of services with increasing complexity.

In order to include a new *device* into the system, the following questions must be answered:

- What information can the *device* provide?
(e.g. *What is the force?*)
- To what commands can the *device* respond?
(e.g. *Go to point (x,y,z).*)
- What events can the *device* detect to issue an interrupt?
(e.g. *Collision!!!*)

3. Experimental Task

We have used APHRODITE to implement an assembly task that involves distributed planning, control and sensing. In this section, we describe the task as well as how APHRODITE was used to integrate the various components.

3.1. *Task description*

In this task, we are trying to insert a plate into a geometrically complex block with a slot, while monitoring the assembly process with force feedback and visual sensors. The plate is held by a robotic arm (Arm1). The position and orientation of the box are determined by visual calibration, and are precise to tolerances on the order of 5mm. A precise model of the object is also available from the CAD server. Since there is inherent uncertainty in the location of the box and plate, the approach trajectory has to be planned so that we 1) determine an appropriate approach vector for the connector and 2) contact forces are monitored to detect contact with the box and to maintain these forces below a desired level (i.e we do not want to impact the box with too great a force upon contact). The task constraints are that we minimize the approach time for the trajectory while not exceeding a specified impact force. The trajectory planner was originally developed as part of another project [15].

The second robotic arm (Arm 2) has a camera attached to its end effector and is used for surveillance. The MVP sensor planning system, which is also part of a separate, ongoing research project [1, 13, 14], is used to plan the approach vector of Arm1 and the sequence of viewpoints used to observe the task. The sensor planner is used to focus the camera on Arm1 and maintain this focus as Arm1 moves, as well as to compute a time-varying viewpoint which maintains a good view of a insertion task.

A breakdown of the task is as follows:

- Navigate Arm2 via teleoperation towards the box using the camera output and the CAD model.
- Plan the angle of approach to the box for Arm1.
- Taking into consideration the uncertainty in object position/orientation (computed on-line), plan the approach trajectory for Arm1 so that the probability of success is optimized.
- Plan a sequence of viewpoints for monitoring the insertion.
- Execute the two plans, i.e. approach the object by Arm1 and perform a compliant insertion using the planned trajectory, while monitoring the insertion using the camera mounted on Arm2 .

Figure 3 shows a snapshot of the task being executed.

3.2. System Architecture

The system consists of a network of general-purpose and special-purpose processors and a dual-arm robot system equipped with force and vision sensors. The assembly task includes sensor planning and spatial reasoning from a CAD model description, trajectory planning, automated visual tracking, and force control. The modules in the system include (see figure 2):

- A geometric modeling system that encodes geometric and topological information about the objects in the robotic workspace, including the robot's themselves.
- The trajectory planning module.
- RCCL, a real-time dual robot control system with a force sensor.
- Real-time, hidden surface display of the geometric model of an object in the workspace that is being monitored.
- The sensor planning system.
- A teleoperated user-interface employing a 6-DOF force measuring joystick (Spaceball).

It is important to stress that each of these modules was originally a stand-alone application that was developed for a variety of different robotic tasks and environments. APHRODITE allowed us to effectively combine these stand-alone modules into a more complex robotic system.

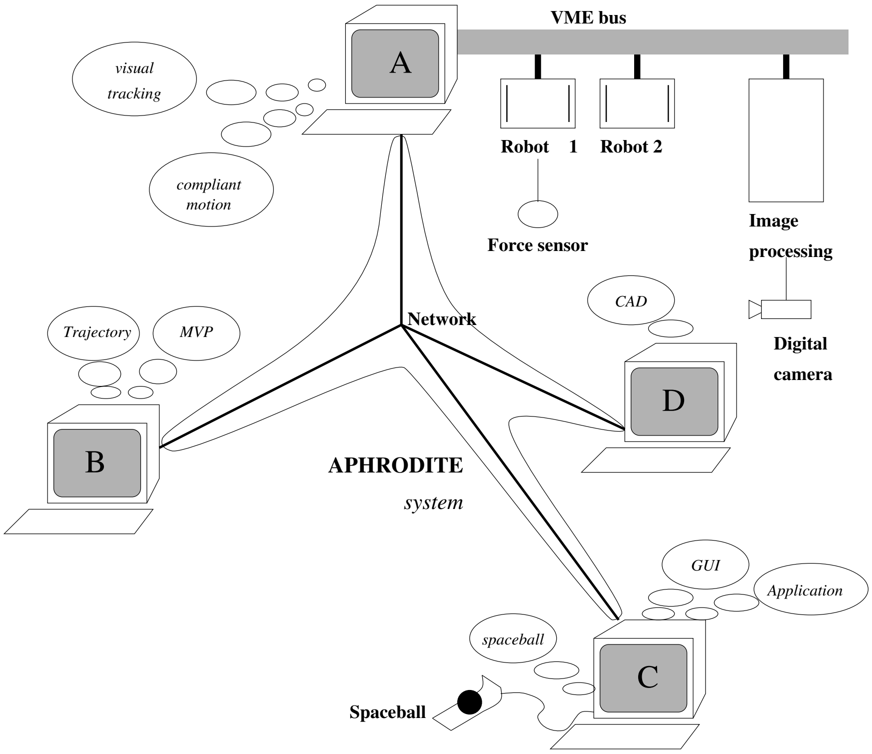

Figure 2: Distributed Robotic System for Planning, Control and Sensing. Hardware modules are denoted in boldface, software in italic.

There are multiple general purpose computers connected by APHRODITE. The robot control *device*[3] (on host A) communicate with two robot controllers through the VME bus. Its task is to provide accurate positioning, as well as compliance during contacts with obstacles. The vision *device* (also on host A) communicate with the parallel image processing engine (PIPE) across the VME bus. The trajectory planning *device* and the MVP sensor planning *device* are run on host B. Host C runs the spaceball *device* as well as the application itself. The CAD *device* runs on host D.

Upon system start-up, the application opens the appropriate devices and begins a dialog process that communicates with the user. The user then executes the phases of the task, step by step.

4. Conclusion

In this paper, we have presented a distributed robot programming system and a task which demonstrates the basic features of that system. Due to the task's complexity and computational demands, it was impractical to implement it on a single processor. APHRODITE allowed a network of processors to be used and the system's architecture enabled the disparate subsystems to coordinate their activities transparently. APHRODITE is based on a distributed architecture that combines the flexibility of a network with a responsiveness of a common-bus system. The overall functionality had to be carefully divided between different modules in the system to exploit the advantages and suppress the maladies of both approaches. APHRODITE makes

[3] Here, we use the term *device* to refer to APHRODITE devices which can be either physical devices or planning modules.

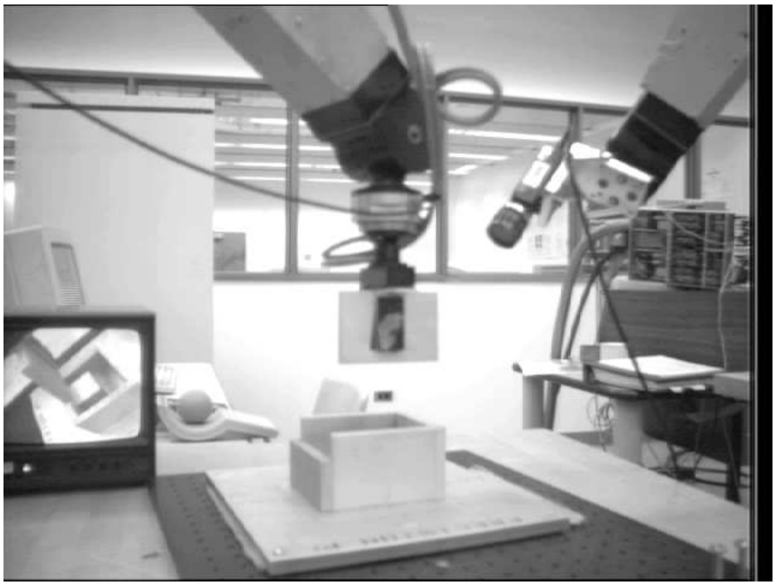

Figure 3: The insertion task in progress. The monitor in the lower left corner shows the image from the surveillance camera held by the Arm 2.

it easier to implement specific software solutions to the planning and sensing problems in a robot system while providing clean, modular interfaces between them. The result is a flexible, general-purpose and cost-effective system capable of controlling very complex robotic systems.

References

[1] Steven Abrams and Peter K. Allen. Sensor planning in an active robotic work cell. In *Proceedings SPIE Intelligent Robotic Systems Conference on Sensor Fusion IV: Control Paradigms and Data Structures*, Boston, MA, November 1991.

[2] C. Geschke. A system for programming and controlling sensor-based robot manipulators. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, PAMI-5(1):1–7, 1983.

[3] S. Hayati, T. Lee, K. Tso, P. Backes, and J. Lloyd. A unified teleoperated-autonomous dual-arm robotic system. *IEEE Control Systems*, 1991.

[4] V. Hayward and R. P. Paul. Robot manipulator control under unix: RCCL, A Robot Control C Library. *Intl. J. of Robotic Res.*, 5(4):94–111, 1986.

[5] J. Lloyd. *Multi-RCCL Reference Manual, Release 4.0 - Multi-robot version*. McGill University, 1988.

[6] T. Lozano-Pérez. Robot programming. *Proceedings of the IEEE*, 71(7), 1983.

[7] D. J. Miller and R. C. Lennox. RIPE: A robot independent programming environment. In *SPIE Vol. 1607: Intelligent Robots and Computer Vision*, pages 518–529, 1991.

[8] Motorola, Inc. *VMEexec User's Guide*, 1990.

[9] C. F. Ruoff. An advanced multitasking robot system. *Industrial Robot*, 1980.

[10] D. Schmitz, R. Hoffman, P. K. Khosla, and T. Kanade. Chimera: A real-time programming environment for manipulator control. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 846–852, 1989.

[11] K. Schwan, P. Gopinath, and W. Bo. CHAOS - kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, 36(8):904–916, 1987.

[12] R. Simmons. Concurrent planning and execution for autonomous robots. *IEEE Control Systems*, pages 46–49, February 1992.

[13] K. Tarabanis. *Sensor Planning and Modeling for machine vision tasks*. PhD thesis, Department of Computer Science, Columbia University, October 1991.

[14] K. Tarabanis, Roger Tsai, and Peter K. Allen. Automated sensor planning for robotic vision tasks. In *IEEE International Conference on Robotics and Automation*, Sacramento, April 9–11 1991.

[15] Aleksandar Timcenko and Peter K. Allen. Modeling uncertainties in robot motions. In *AAAI Workshop on Real-World Autonomous Robots*, Cambridge, MA, November 1992.

[16] L. Whitcomb and D. Koditschek. Robot control in a message passing environment: Theoretical questions and preliminary experiments. In *IEEE International conference on Robotics and Automation*, pages 1198–1203, 1990.

[17] Wind River Systems, Inc. *VxWorks Reference Manual*.