REAL-TIME SOFTWARE FOR ROBOTICS

Ingemar J. Cox, David A. Kapilow, Walter J. Kropfl, and Jonathan E. Shopiro

Ingemar J. Cox, David A. Kapilow, Walter J. Kropfl, *and* ***Jonathan A. Shopiro*** *are members of technical staff in the Robotics Principles Research Department at AT&T Bell Laboratories, Murray Hill, New Jersey. Mr. Cox's current research interests are in real-time software, computer vision, and autonomous robot vehicles. He has a B.Sc. in electronic engineering and computer science from University College, London, and a D.Phil. in engineering science from the University of Oxford. He joined AT&T in 1984. Mr. Kapilow's current research is in real-time operating systems and environments. He has a B.A. in physics from Dartmouth College and an M.S. in computer science from Rutgers University. He joined AT&T in 1980. Mr. Kropfl's work has centered on a variety of robotics control problems. He received a B.S.E.E. from Pennsyl-*
(continued on page 72)

This paper describes two components of real-time software for robotics: real-time operating systems and the systems programming language. In particular, the design issues related to real-time operating systems are discussed and our own system, derived from the UNIX® operating system, is described. No single real-time operating system is appropriate under all circumstances. However, it is shown that at least some continuity in the development environment can be provided. We have chosen to use a general-purpose programming language, C++, which has several advantages for real-time robot programming. In particular, it is shown how facilities within C++ can be used to guarantee initialization and proper termination of hardware subsystems. Support for concurrency is considered important, especially as robot systems become more complex. Proper support for concurrency can simplify communication and synchronization within a robot system. Two methods of providing concurrency within C++ are discussed.

Introduction

Modern manufacturing requires assembly machines and processes to be controlled by computers. Programmable control of numerical control machines, robots, and manufacturing processes provides tremendous flexibility in the manufacturing systems. This paper is concerned with the software needed to control such machines and, in particular, robots.

Real-time software for robotics must deal efficiently with devices and externally generated events under rigid timing constraints. These timing constraints may vary from a few microseconds in a short-time-resolution robot control loop to several seconds in other applications. The real-time nature of the software imposes constraints both on the underlying operating system and on the programming languages used. A subsequent section discusses the requirements of a real-time operating system and how these requirements can be met for a variety of timing and hardware constraints. Our own requirements have been met by an internally developed operating system derived from the UNIX system.

At the language level, there are perhaps three main classes of robot programmer: the operator, the application developer, and the systems programmer.[1] The operator typically has little programming experience, but "programs" the machine via a user-friendly menu-driven graphical interface or teach pendant. The operator is not concerned with meeting strict timing constraints but only in programming the robot to perform a desired sequence of actions. The application programmer, responsible for each new application, may use a specialized language to list the motions and operations the robot will perform. The application developer may have restrictions on how quickly a sequence of operations must be performed in order to meet required manufacturing rates. These constraints are significantly longer than those at the systems programming level. The systems programmer is responsible for the development of both the application-level environment and, most importantly, the underlying code that drives the robot. The systems programmer is concerned with the detailed timing constraints of the robot system, constraints which may be as short as several tens of microseconds.

This paper also discusses systems programming languages for robotics and why we chose to use a general-purpose programming language, C++.[2] As robots become more complex, the need to provide support for concurrency, synchronization, and communication both within and between robot systems becomes paramount. The discussion includes the topic of concurrency and considers two alternative approaches to how concurrency can be provided within C++.

Real-Time Operating Systems

Real-time operating systems provide support for the development and execution of application programs, which must meet timing constraints imposed by the devices controlled. A real-time operating system must balance the need to meet these timing constraints with other issues such as the ease of software development, the application programmer's software expertise, the architecture and speed of the underlying computer hardware, and the implementation cost.

This section examines the impact of some of these issues on real-time operating system design and describes the environment we have found useful for our work.

Design Issues. Since timing constraints may vary from a few microseconds to several seconds, it is not surprising that no single real-time operating system is best in all situations. As an application's timing constraints become tighter, compromises in the real-time operating systems support services usually have to be made because less overhead is tolerable.

The most stringently constrained real-time applications may be possible only if there is minimal interference from the operating system. In these applications, often the simplest solution is to not have any operating system. Since the application has complete control over the processor, its performance is limited only by the hardware. The disadvantage of this approach is that considerable expertise is required by the programmer, especially as the complexity of the application increases.

When there is no underlying operating system, a separate host computer is needed for program development. The application is edited and cross-compiled on the development system, and then either a hardware emulator is used or the binary program is loaded onto the satellite real-time computer via a communications link. Separation of the development computer from the the real-time controller allows the controller hardware and software to be simpler, since the needs of the development system can be satisfied on the host. However, the need for a separate development computer and the associated communications link adds to the overall cost, especially for low-end systems.

When an operating system is appropriate, real-time executives provide the next level of support. These offer multitasking, priority-based preemptive scheduling, and simple but efficient interprocess communication and synchronization calls. Interrupt latencies are increased over the raw processor times since interrupts must be masked to avoid context switches during critical sections of code. The executives and applications may be placed in read-only memories (ROMs) for embedded systems. Separate host computers are usually used for program development. Context switching times for these systems are usually below 100 μs.

At a still higher level, stand-alone real-time systems provide both the development and execution environment on the same processor. However, the corresponding operating systems are inevitably more complex, since they must provide the additional development support. In addition, if the development-related hardware and software become active during the real-time program's execution, they can have a detrimental effect on the system's real-time performance.

In both stand-alone and real-time executives, memory management may be used to provide better program development environments, albeit at the expense of longer context switching times and additional memory access wait states. The input and output (I/O) may be done directly by the application, or it can be provided by the operating system through system calls. The former yields better performance, while the latter leads to a friendlier execution and development environment.

In both I/O and computationally intensive real-time applications, multiple processors may be required. These may be either tightly coupled (sharing memory) or loosely coupled. It is not uncommon to have both in the same system. Data traffic between the processors may have simple or complex paths. A general, uniform communications mechanism between all processors yields a superior programmer interface,[3] while simple specialized paths can be implemented more efficiently.

NRTX. Our own real-time systems use NRTX (New Real-Time Executive),[4] a locally developed real-time executive derived from the UNIX operating system. NRTX is capable of supporting a wide variety of hardware with sub-millisecond response. The decision to write our own executive was largely due to a desire to replace the software associated with a vendor-supplied hardware device with a UNIX-system-based executive that was not available commercially. We had several motivations for adopting the UNIX system as a base: familiarity, preference to deal with a single system, unwillingness to give up accustomed software development tools, and availability of large bodies of code that could be ported from UNIX systems to real-time controllers. Adopting the UNIX system base has yielded other benefits. For example, access to the source has made it possible to tune the operating system primitives (basic instructions) to the applications.

The NRTX executive is primarily derived from the Version 7 UNIX system kernel and an internal real-time executive called RTX. This particular version of the UNIX system was chosen because of its small size and simplicity. Essentially, the file system was removed, the scheduling algorithm was modified to a priority-based preemptive one, and the synchronization mechanisms available to device drivers were made accessible to application code. Additional system calls were added for simple interprocess communication. Device drivers can be placed in the executive and accessed through system calls, or, alternatively, applications can handle I/O directly and service their own interrupts. Executive system calls can be preempted for low context switching latency. Application processes run in the supervisor mode so that the processor's hardware priority can be quickly modified. Memory management is not used, therefore processes can share memory.

Hardware requirements for NRTX are minimal. A processor, 64 kilobytes of random-access memory (RAM), a periodic timer, and a communications channel to the UNIX development system are all that is required. A typical executive including a local-area-network driver, a serial line driver, and application code for the downloading protocol requires 23 kilobytes of text address space and 9 kilobytes of data, allowing NRTX to be placed in ROM if desired. In addition, NRTX now runs on all the members of the Motorola 68000 family as well as on the AT&T WE® 32100 microprocessor. Portability is largely inherited from its UNIX system parent; 95 percent of the executive is written in C language,[5] with the rest in assembly language. The nonportable sections of code deal mainly with context switching and the treatment of traps and interrupts.

As an illustration of the speed of the system, consider the time taken to send a message between two

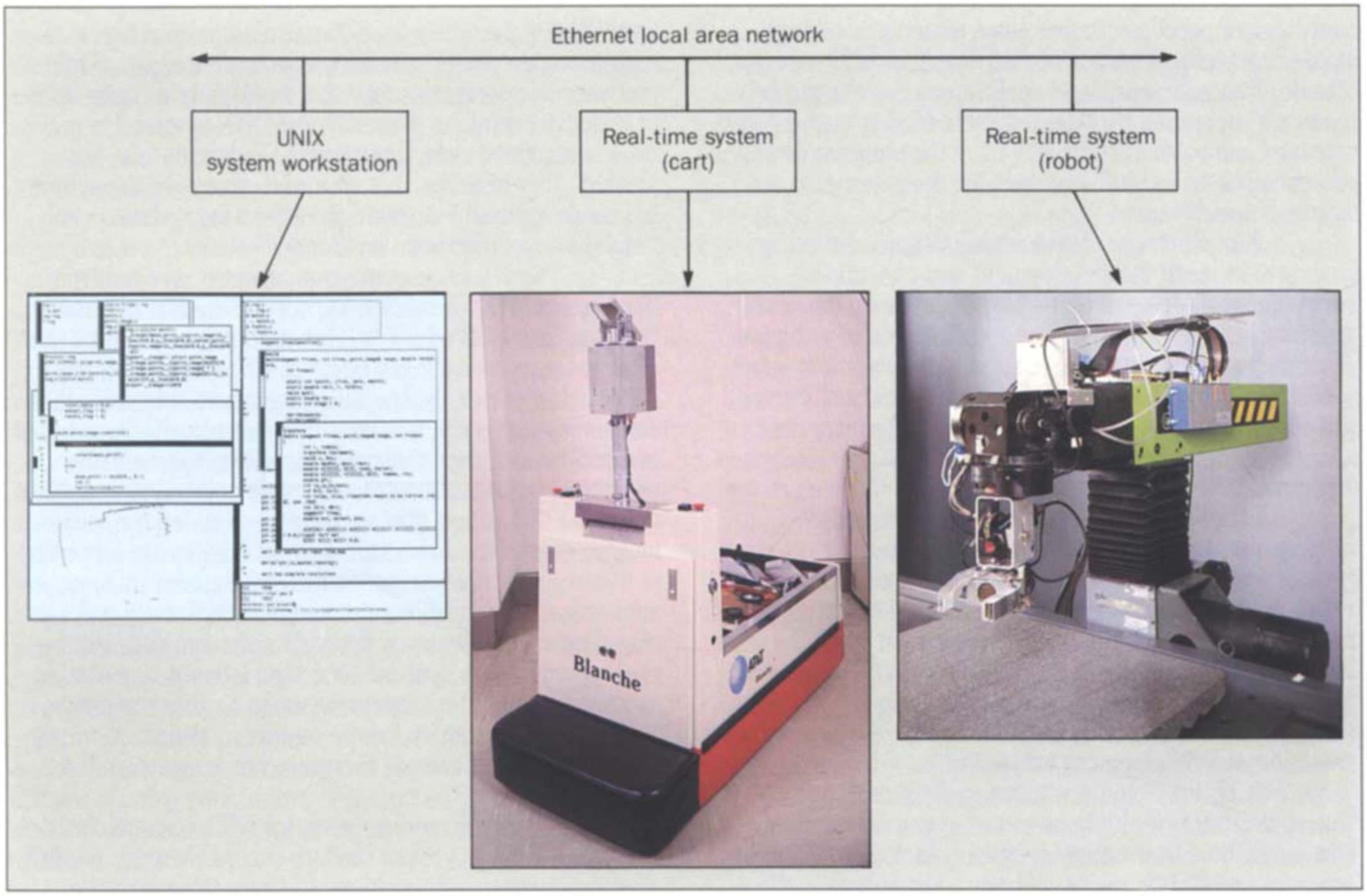

Figure 1. Hardware environment.

processes. The time to send a 20-byte message buffered by the system, do a context switch, and receive the message is 379 μs on a 16.67-MHz 68020 microprocessor with 1.5 wait states and 926 μs on a 12.5-MHz 68010 with no wait states. These are the most complicated system calls in the executive.

Support libraries allow application processes to use UDP/IP protocols over the local area network (LAN). A package built on top of this allows programs to use C language standard I/O library functions to access the file systems of UNIX system machines on the LAN. The downloading communications software is largely derived from the AT&T 5620 dot-mapped display terminal.[6]

The hardware environment used for our real-time program development is shown in Figure 1.

Host UNIX system workstations are used to develop application programs. The binary files created are then downloaded to dedicated real-time controllers via a local area network. The UNIX system workstations support high-resolution bit-mapped graphics displays and multiple windows. The bit-mapped display of Figure 1 illustrates the advantages of such an environment. The display includes a window for an editor modifying a program under

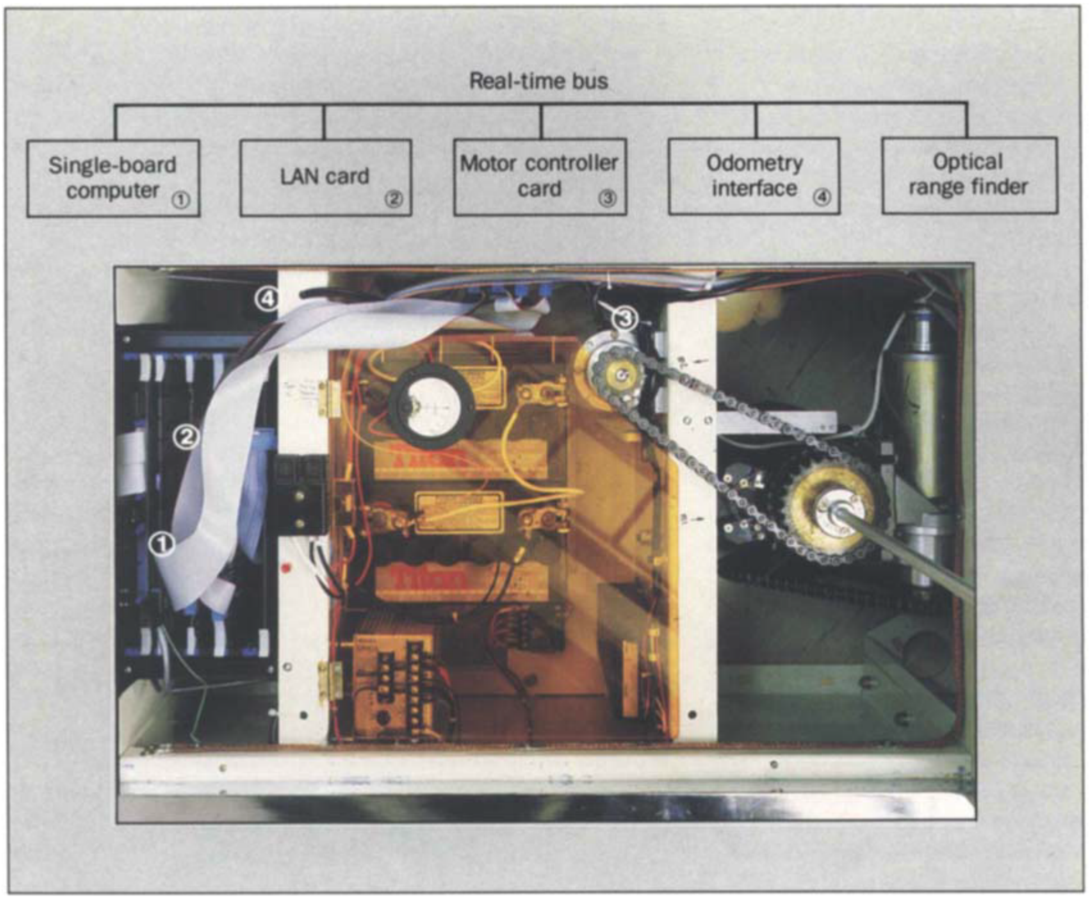

Figure 2. Uniprocessor real-time system (autonomous cart).

test, a terminal window on the UNIX system machine, a graphics window that continuously displays optical range data being acquired by the cart controller in Figure 2, and a mouse-based debugger used to remotely debug a C++ real-time application.

The real-time controllers have been constructed from single-board computers on industry-standard buses. Connections to real-time devices and sensors have been made with a combination of commercially available interface cards and, when necessary, custom interface designs. The LAN interface, which is currently the Xerox Corp. Ethernet™ network, provides a high-bandwidth connection to the UNIX development systems and allows communications between real-time controllers. The Ethernet network was chosen because it presently provides the best balance of reliability, performance, and cost for our applications.

Figure 2 shows the hardware of a typical uniprocessor real-time system. It is the controller for an autonomous cart used in research in navigation and sensing.[7] It contains a single Motorola 68020 processor board with 256 kilobytes of random-access memory, a motor controller interface, multiple sensor interfaces, and an Ethernet interface. The Ethernet interface is for programmer convenience during program development and may be disconnected during cart operation.

Figure 3. Multiprocessor real-time system (robot).

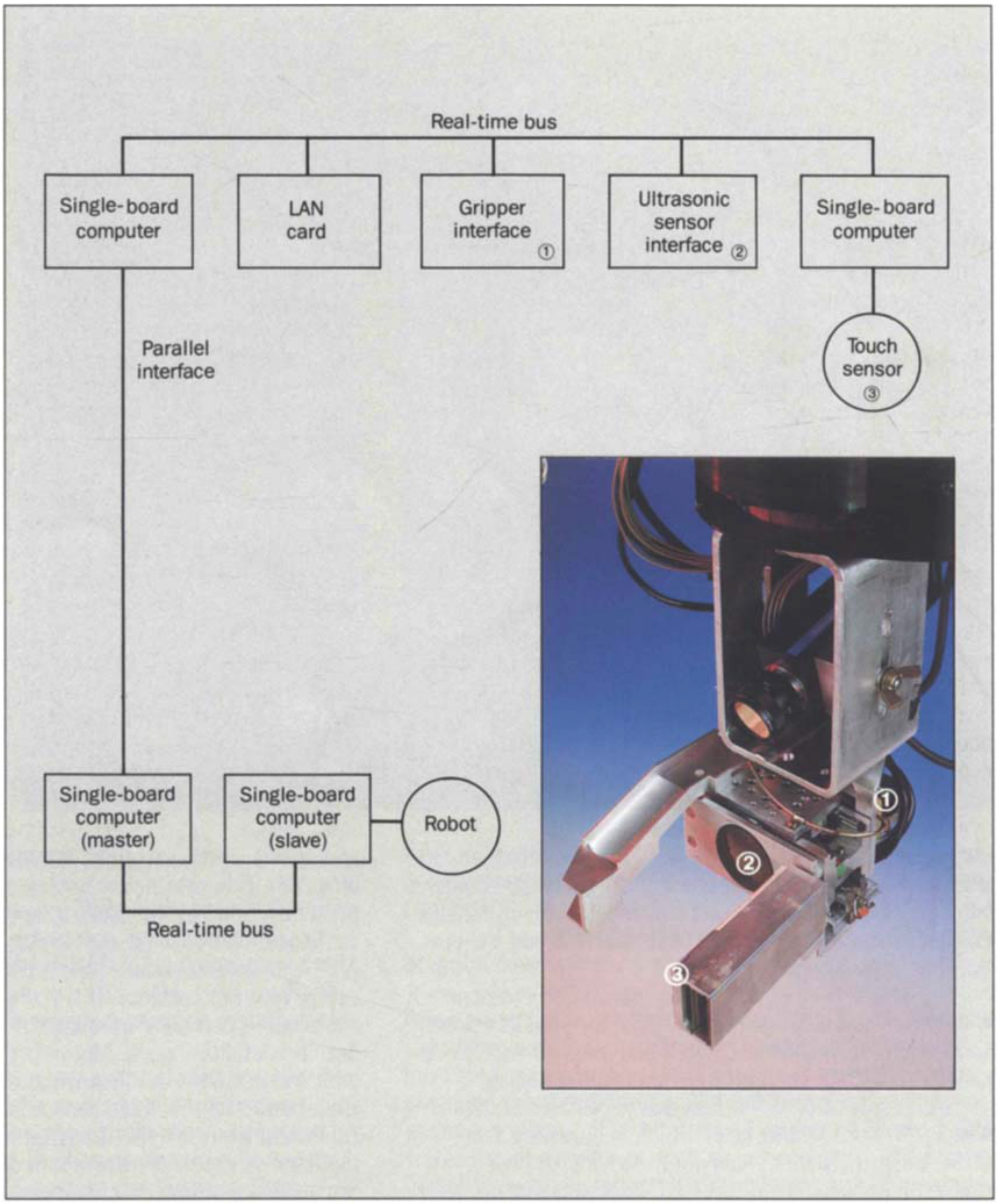

Figure 3 shows a multiprocessor real-time controller. This is the controller for a commercial Cartesian-coordinate robot enhanced with a controlled-compliance gripper[8] and ultrasonic and touch sensors. Multiple processors are used because of the large computational requirements of the components. Several of the processors are dedicated to particular tasks. One processor controls the robot's servomotors in a feedback loop, while another collects and analyzes data from the touch sensor array on the gripper. The interconnections between the processors are partially determined by vendor-supplied hardware. This system is currently being used as one component in speech-controlled robot research.[9]

The slave processor connected to the robot in Figure 3 does not use any real-time operating system support. Instead the entire program on this processor is one large interrupt loop that executes periodically. Any interference from the operating system would adversely affect its operation. The remaining processors in Figures 2 and 3 run NRTX.

For many less tightly constrained real-time applications, it becomes desirable to use an operating system that allows the tradeoff of more support services at the expense of higher overhead and complexity. Even though the NRTX application environment is similar to the UNIX system, it is not close enough. For example, making even minor modifations in large sections of non-real-time code becomes annoying over time.

Work is therefore in progress to create a complete stand-alone UNIX system with real-time benchmarks based on the Ninth Edition kernel. Preliminary real-time scheduling enhancements made on a 16-MHz 68020 bus board look promising. Up to 14 real-time tasks can run concurrently with context switches on the order of 400 μs, while maintaining a full UNIX system environment. For many applications the increased overhead of the operating system will be compensated by support services such as a mouse-based multiprocess debugger[10] and networking that is transparent to the application.

Robot Systems Programming

There are three primary levels of robot/machine programmer:

- The end user or operator
- The application programmer
- The systems programmer.

The operator, responsible for control of the machine on the factory floor, may have little or no programming experience. Consequently, the operator interface may not resemble a conventional programming language at all. Most likely, it will be a user-friendly, menu-driven graphical interface or a *teach pendant*, which is moved and whose motion is recorded so as to teach a robot a task. The application programmer, responsible for each new application, may use a specialized language to list the motions and operations the robot will perform. An application-level language is usually interpreted; interpreters provide a more friendly environment in which to develop and debug small programs. Finally, the systems programmer is responsible for the development of both the application-level environment and, most importantly, the underlying code that drives the robot. We are concerned with the systems-level code where the timing contraints, which may range from tens of microseconds, significantly affect the program code.

A robot or machine can be considered to be like any other peripheral device, but with certain special characteristics. In many robot applications, however, robot-specific operations constitute only a small fraction of the total program.[11] The remaining operations are typically concerned with operator interfaces, database access, and numerical computation. Consequently, we believe that an appropriate general-purpose programming language should be used for writing robot programs. This view is supported by the designers of the robot programming language AML.[1] In fact, the use of a general-purpose language for robot control is not new. A common language used for robot programming is a BASIC superset. An extension of Pascal has also been used for robot programming.[12] Moreover, we have been using the C language for this purpose for several years now, as have others.[13] In fact, C appears to be becoming the systems programming language of choice among robot manufacturers. We are currently using C++, a superset of the C language, the advantages of which are described in the following section.

Many current robot systems exhibit a significant degree of concurrency. In addition, there is a need to coordinate a robot's activities with those of other robots and with external sensory events, as well as to provide high-level communication facilities between multiple robots and other devices. Conventional programming environments do not adequately support these requirements. However, this is not sufficient reason to advocate a special-purpose robot programming language. On the contrary, we believe that a general-purpose language with appropriate concurrent programming facilities would be well-suited for programming robots. Concurrent programming simplifies writing robot programs by allowing the direct expression of a robot's concurrent activities and by simplifying the tasks of synchronization and communication.[14] We believe concurrent programming is important to robotics, especially as systems become more complex. Later in this article, we discuss issues related to concurrent programming and briefly describe two approaches for providing concurrent programming within C++.

C++ for Robotics. Within the Robotics Principles Research Department, the C++ programming language is used extensively. The reason for this relates to the following characteristics of C++:

"The key concept in C++ is class *. . . Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user-defined types, dynamic typing, user-controlled memory management, and mechanisms for overloading operators. C++ provides . . . facilities for type checking . . . expressing modularity . . . symbolic constants, inline substitution of functions, default function arguments, overloaded function names, free store management operators, and a reference type. C++'s ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.) . . . allows the user-defined types to be implemented with a pleasing degree of efficiency."*[2]

The strong data typing facilities provided by C++ allow many of the most common programming errors to be detected at compile time. This is especially important for real-time applications in which debugging facilities are often primitive or the use of a debugger causes a violation of the real-time constraints. The compiled code is very efficient with little overhead being paid for the high-level facilities provided. In addition, the in-line function declarations allow further refinement of the run-time code. Languages and facilities for both the application and the operator levels can be built on top of C++. Further, C++ can be extended (through data abstraction and operator overloading) to allow the direct expression and manipulation of vectors, homogeneous transformations and other robot and machine-specific data structures.

Data abstraction. The use of data abstraction as a means for improving software quality is well-known, and its applicability to robotics has been previously discussed.[15] Briefly, data abstraction provides support for:

- *A well-defined user interface.* There is a clearly defined set of operations associated with each abstract data type.
- *Data hiding.* The details of the implementation can be hidden from the user.

Data abstraction is useful in robot systems programming. A class definition can embody the functional specification of each physical subsystem while hiding the implementation details. For example, in the case of an autonomous robot vehicle, we might have four distinct low-level subsystems: odometry (for position estimation), rangefinder (for environment sensing), steering, and drive. Each of these subsystems is represented by a class with a small well-defined set of user operations. In the case of the drive motor we have:

```
class drive_motor
{
 // private implementation details
public:
 double speed(double);
};
```

The member function `speed` commands the vehicle to travel at the prescribed speed.

Representing each robot subsystem as an abstract data type has advantages. Returning to the `drive_motor` example, the current implementation uses a custom board built around a Hewlett-Packard HCTL-1000 motion-control chip,[16] while a previous implementation used a Galil dual-axis servo-controller board. These boards have very different means of control. The HP board uses 16-bit binary control words to configure and control it; the Galil board expects ASCII character strings. While the software to control the drive motor can be very different, depending on the hardware used, users of the `drive_motor` class are unaware of it, provided the public member function `speed` remains supported. In this manner, generic subsystems and entire systems can be developed. In fact, it is quite possible to create a generic form of robot with well-defined motion commands that is relatively independent of the physical structure of the robot, be it Cartesian, Scara, etc. The possibility of writing robot-independent code is an interesting example of the advantages provided by a language that supports data abstraction.

Initialization and self-test. The first lines of a robot program typically call initialization routines. This is error-prone. Forgetting to call initialization routines is not uncommon. C++ provides a constructor mechanism associated with a class which both guarantees initialization and hides it from the user.[17]

For example, a `drive_motor` always needs to be initialized on power up, and motion-control boards need to be configured with pole, zero, and gain parameters. We can ensure that this initialization is carried out by use of the associated constructor:

```
class drive_motor
{
  // private implementation details
public:
  drive_motor(); // constructor
  double speed(double);
};
drive_motor::drive_motor()
{
  // initialization code
}
```

The `drive_motor` constructor is executed as soon as a `drive_motor` object is instantiated. It is completely transparent to the user and guarantees initialization of the associated subsystem.

While in the constructor, it is also advantageous to confirm the operational status of the subsystem. Self-test is important. It improves both the safety of the system (for example, it can be dangerous to move a vehicle if the steering is malfunctioning) and the maintenance of the system (error diagnostics printed by the subsystem's self-test procedures help in the identification of a failure and its subsequent repair). If all subsystems successfully perform their self-tests, then the user is guaranteed that the system is fully functional. This is advantageous; it is frustrating to spend time debugging a program only to find that an underlying subsystem failure is at fault.

It should be noted that the degree to which a subsystem can be tested varies greatly. For example, it is possible to guarantee completely the operational status of the `rangefinder`; tests can be performed to ensure that the ranger is scanning and that the corresponding range values are sensible. On the other hand, very little can be said of the operational status of the `drive_motor` or `odometry` subsystems without moving the vehicle. However, uncommanded motions (initiated by the constructor) are undesirable from a user-programming perspective and such tests are therefore not performed.

Termination and fail-safe operation. Correct termination of a robot program is as important as correct initialization. It is imperative that the robot system be left in a safe state upon completion of the program. Just as with initialization, one would like to guarantee fail-safe termination transparent to the user, even if the program is aborted prematurely. Therefore C++ provides destructors, complementary to constructors, that can be used to guarantee such fail-safe operation.[17] Destructors are invoked when the instantiated object goes out of scope or when the function `exit` is called.

Returning to the `drive_motor` example, we have

```
class drive_motor
{
  // private implementation details
public:
  drive_motor();
```

```
    // constructor
  ~drive_motor();
    // destructor
  double speed(double)
};

drive_motor::~drive_motor()
{
    // termination and failsafe code
}
```

The destructor `drive_motor` ensures that the vehicle is brought to a halt before the program terminates.

Concurrency and C++. A convenient way to structure a robot control program is as a set of processes, each of which controls a particular robot subsystem. These processes must communicate with each other and wait for asynchronous events in the environment. This requires a programming environment that supports *concurrency* (see Panel 1). Using concurrent programming for robotics has received previous attention. For example, Concurrent Pascal has been used for industrial systems programming,[19] although not specifically for robotics. Kanayama[20] discusses the need for concurrency in relation to a mobile robot. In addition, several commercial robot languages support some form of concurrency.[21] Most previous work has addressed some of the issues of synchronization, communication, and process creation and termination. However, unified support for all these issues was sorely lacking. Consequently, we have investigated providing and using concurrent programming within C++.

Support for concurrent programming has been added to C++ in two ways. In Concurrent C,[22,23] processes communicate by means of synchronous transactions. The syntax of Concurrent C provides facilities for declaring and creating processes, process synchronization and interaction, process termination and abortion, and priority specification and waiting for multiple events, among other things. Concurrent C, like the Ada language,[24] is based on an extension of the rendezvous concept called the *extended rendezvous* or *transaction.* The transaction allows bidirectional information transfer (communication) during the rendezvous. After a transaction has been established, the process requesting service is automatically forced to wait until the server completes the requested transaction (synchronization); the transaction results are then sent back to the waiting client.

Panel 1. Robot Program Needs

Robot programs are characterized by a need to:

- *Deal with concurrent activities.* A robot program must concurrently control each of the robot's joints and sensors, and the part feeders and conveyers associated with a robot workcell.
- *Synchronize actions with external events.* A robot must frequently wait for an event before executing an action.
- *Communicate with other robots and processes.* Communication with other robots and factory processes is essential to avoid the robot becoming an "island of automation." In fact, it has been observed[18] that in the design of a program involving two cooperating robots, up to 30 percent of the code was concerned with interaction between the two robots.
- *Handle interrupts on a timely basis.* The language must allow the user to supply associated interrupt service routines.

The other concurrent programming system for C++ is called the task library.[25] Unlike Concurrent C, it uses no additional syntax, but instead relies on the constructors and function invocations already in C++. Thus, it does not require any compiler modifications. Also, the task library provides lower-level communication and synchronization facilities than Concurrent C, relying on the programmer to build customized interfaces using the abstract data types of C++. Communication between processes in the task library is through a passive intermediate object, rather than directly through a transaction as in Concurrent C. That is, when a process does an operation on such an object, if the object is not in a ready state, the process waits until some other process or interrupt makes it ready. Each class of object can have its own operation and readiness criteria. For example, a queue head object provides the *get* operation, and it is ready if there is anything in the queue; a semaphore object provides the *wait* operation, and it is ready if previous *signal* operations (which always succeed immediately) outnumber previous waits.

These two architectures for concurrency, though different, are functionally equivalent, in that any program that can be written using one can be written using the other. A syntactic difference between the task package and Concurrent C is that the task package is a library, consisting of routines that are called as functions at run time, and has no support in the compiler, while Concurrent C expresses concurrency with new syntactic structures and requires a modified compiler. Although the question of whether or not the programming language itself should provide concurrent programming facilities remains open, it is clear that concurrent programming facilities are an important component of a programming system for robotics.

Conclusions

No single real-time operating system is capable of meeting the entire range of requirements that user applications may require. Indeed, the severest timing constraints may preclude the use of *any* operating system. However, one can still attempt to provide continuity in the programming environment. For example, a UNIX host can still be used to develop and cross-compile code for both a bare hardware system and satellite real-time operating systems. If the different real-time operating systems contain the same primitives, code can easily be transported between them.

Our work has focused on providing such a continuity of environment with continued effort directed at a complete UNIX system with real-time enhancements. A strong motivation for this work is to gain access to readily available software for transparent networking and multiprocess debugging. Such networking capability is extremely important, particularly at the robot workcell level and above where it may be necessary to tie many components together. An additional potential benefit of a complete, stand-alone, real-time UNIX system is that the development machine and the target machine can become one and the same. This both reduces hardware proliferation and offers enhanced reliability in some circumstances, in that local machines or whole areas can more readily operate independently.

Work is continuing on concurrent programming as well as on investigating systems level support for the application and operator levels. In this connection, C++ (and its concurrent programming enhancements) has proved to be a powerful language in which to program real-time robotic applications. The potential for writing robot independent code through the use of generic robot classes is exciting and may offer the opportunity for significantly reducing the cost of robot and machine-control software development.

Acknowledgments

The authors are pleased to acknowledge valuable discussions with N. H. Gehani.

References

1. L. R. Nackman et al., "AML/X: A Programming Language for Design and Manufacturing," *Proceedings, Fall Joint Computer Conference*, Dallas, 1986.
2. B. Stroustrup, *The C++ Programming Language*, Addison Wesley, Reading, Massachusetts, 1986.
3. R. D. Gaglianello and H. P. Katseff, "MEGLOS: An Operating System for a Multiprocessor Environment," *Proceedings, IEEE Intl. Conf. on Distributed Computing Systems*, Denver, 1985, pp. 35-42.
4. D. A. Kapilow, "Real-Time Programming in a UNIX Environment," Symposium on Factory Automation and Robotics, New York University, Sept. 9-11, 1985.
5. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
6. R. Pike, "The Blit: A Multiplexed Graph Terminal," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Part 2—Computing Science and Systems, October 1984.
7. I. J. Cox, "Blanche: An Autonomous Vehicle for Structured Environments," *Proceedings, IEEE Intl. Conf. on Robotics and Automation*, April 1988.
8. M. K. Brown, "Controlled Impedance Robot Gripper," *AT&T Bell Laboratories Technical Journal*, Vol. 64, No. 4, April 1985.
9. M. K. Brown, "Controlling a Robot With Sonic and Ultrasonic Means," *Proceedings, IEEE Ultrasonics Symposium*, Denver, 1987.
10. T. A. Cargill, "The Feel of Pi," *Proceedings, Winter USENIX Meeting*, Denver, January 1986.
11. R. H. Taylor, P. D. Summers, and J. M. Meyer, "AML: A Manufacturing Language," *Intl. Journal Robotics Research*, Vol. 1, No. 3, 1982, pp. 19-41.

12. C. Blum and W. Jakob, *Programming Languages for Industrial Robots*, Springer-Verlag, New York, 1986.
13. V. Hayward and R. P. Paul, "Robot Manipulator Control Using the C Language under UNIX," *Proceedings, IEEE Workshop on Languages for Automation*, 1983, pp. 3-10.
14. I. J. Cox and N. H. Gehani, "Concurrency and Robotics," *Intl. Journal of Robotics Research*, to be published.
15. R. A. Volz, T. N. Mudge and D. A. Gal, "Using Ada as a Programming Language for Robot-Based Manufacturing Cells," *IEEE Trans. Systems, Man and Cybernetics*, Vol. SMC-14, No. 6, 1984, pp. 863-878.
16. "General Purpose Motion Control IC HCTL-1000," Hewlett-Packard Technical Data Sheet, 1985.
17. I. J. Cox, "C++ Language Support for Guaranteed Initialization, Failsafe Termination and Error Recovery in Robotics," *Proceedings, IEEE Intl. Conf. on Robotics and Automation*, April 1988.
18. R. A. Volz, "Distributed Systems Integration Tools for Robotics," *Proceedings, 1985 Symposium on Factory Automation and Robotics*, Courant Institute of Mathematical Sciences, New York University, 1985, pp. 23-25.
19. B. G. Mortensen, "Use of Concurrent Pascal in Industrial Systems Programming," *Microprocessing and Microprogramming*, Vol. 14, 1984, pp. 155-159.
20. Y. Kanayama, "Concurrent Programming and Intelligent Robots," *Proceedings, Intl. Joint Conf. on Artificial Intelligence*, 1983, pp. 834-838.
21. S. Bonner and K. G. Shin, "Comparative Study of Robot Languages," *Computer*, Vol. 15, No. 12, 1982, pp. 82-96.
22. N. H. Gehani and W. D. Roome, "Concurrent C," *Software—Practice and Experience*, Vol. 16, No. 9, 1986, pp. 821-844.
23. I. J. Cox and N. H. Gehani, "Concurrent C and Robotics," *IEEE Intl. Conf. on Robotics and Automation*, 1987, pp. 1463-1468.
24. U.S. Department of Defense, *Reference Manual for the Ada Programming Language*, July 1982.
25. J. E. Shopiro, "Extending the Task System for Real-Time Control," *Proceedings, USENIX C++ Workshop*, 1987, Usenix Conference Office, Sunset Beach, Calif.

Biographies (continued)

vania State University and an M.S.E.E. from Stevens Institute of Technology. He joined AT&T in 1966. Mr. Shopiro has used the C++ language in databases, interactive graphics, and, currently, robotics.

(Manuscript received October 29, 1987)